

MBEES 2014
MBEES 2014
MBEES 2014
MBEES 2014
MBEES 2014

Tagungsband des Dagstuhl-Workshops

Modellbasierte Entwicklung eingebetteter Systeme X

Holger Giese
Michaela Huhn
Jan Philipps
Bernhard Schätz



VALIDAS



Tagungsband

**Dagstuhl-Workshop MBEES:
Modellbasierte Entwicklung
eingebetteter Systeme X**

**Model-Based Development of Embedded Systems
05.03.2014 – 07.03.2014**

**fortiss GmbH
Guerickestr. 25
80805 München**

Organisationskomitee

Prof. Dr. Holger Giese, Hasso-Plattner-Institut an der Universität Potsdam

Dr. Michaela Huhn, TU Clausthal

Jan Philipps, Validas AG

Dr. Bernhard Schätz, fortiss GmbH

Programmkomitee

Dr. Mirko Conrad, The Mathworks GmbH

Prof. Dr. Ulrich Epple, RWTH Aachen

Prof. Dr. Holger Giese, Hasso-Plattner-Institut an der Universität Potsdam

Dr. Michaela Huhn, TU Clausthal

Dr. Hardi Hungar, DLR

Prof. Dr.-Ing. Klaus D. Müller-Glaser, Karlsruher Institut für Technologie KIT

Ulrich Nickel, Delta Energy Systems

Prof. Dr. Oliver Niggemann, Fraunhofer IOSB-INA

Jan Philipps, Validas AG

Dr. Ralf Pinger, Siemens AG

Prof. Dr.-Ing.habil. Matthias Riebisch, Universität Hamburg

Prof. Dr. Bernhard Rumpe, RWTH Aachen

Dr. Bernhard Schätz, fortiss GmbH

Prof. Dr. Albert Zündorf, Universität Kassel

Inhaltsverzeichnis

MDA für Altsysteme - Ein langer Weg <i>Michael Erskine, Alexander Schäffer and Rico Lieback</i>	1
Towards Symbolic Causality Checking using SAT-Solving <i>Adrian Beer, Uwe Kuehne, Florian Leitner-Fischer and Stefan Leue</i>	11
Die Ergänzung des AUTOSAR Standards für eine durchgängige modellbasierte automobile Steuergeräteentwicklung <i>Jan Meyer</i>	21
Eine Anleitung zur Entwicklung von Simulink-Targets für die Lehre <i>Frank Traenkle</i>	31
Software Design Means for Digital Passenger Cars <i>Ulrich Freund</i>	38
On the right Degree of static Model Analysis for ISO 26262 <i>Heiko Doerr and Ingo Stuermer</i>	47
Effort and Efficacy of Tool Classification and Qualification <i>Mirko Conrad and Ines Fey</i>	52
Model-based Parallelization and Optimization of an Industrial Control Code <i>Ralf Jahr, Martin Frieb, Mike Gerdes and Theo Ungerer</i>	63
Preliminary Experience of using mbeddr for Developing Embedded Software <i>Markus Voelter</i>	73
Towards a Curriculum for Model-Based Engineering of Embedded Systems <i>Bernd-Holger Schlingloff</i>	83
On the Role of Models in Engineering Cyber-Physical Systems <i>Bernhard Schaetz</i>	91

Modellbasierte Konsistenzprüfung von Produktlinien von Automatisierungssystemen <i>Matthias Riebisch and Yibo Wang</i>	97
Regelbasiertes Engineering mithilfe deklarativer Graphabfragen <i>Sten Grüner and Ulrich Epple</i>	103
Assuring Standard Conformance of Partial Interfaces <i>Hardi Hungar</i>	112



Innerhalb der Gesellschaft für Informatik e.V. (GI) befasst sich eine große Anzahl von Fachgruppen explizit mit der Modellierung von Software- bzw. Informationssystemen. Der erst neu gegründete Querschnittsfachausschuss Modellierung der GI bietet den Mitgliedern dieser Fachgruppen der GI - wie auch nicht organisierten Wissenschaftlern und Praktikern - ein Forum, um gemeinsam aktuelle und zukünftige Themen der Modellierungsforschung zu erörtern und den gegenseitigen Erfahrungsaustausch zu stimulieren.



Die Arbeitsgruppe Grundlagen der Informatik am Institut für Informatik der TU Clausthal forscht auf dem Gebiet der formalen Methoden für sicherheitskritische, softwaregesteuerte Systeme. Dies umfasst modellbasierte Entwurfsmethoden sowie formale Validierung und Verifikation. Erarbeitet werden Methoden und Werkzeuge zum Nachweis der funktionalen Korrektheit, aber auch zum Echtzeit- und Störungsverhalten komplexer eingebetteter Systeme. In diesem Rahmen wird auch die Qualitätsplanung und Sicherheitsnachweisführung thematisiert. Die Anwendbarkeit der Lösungen wird immer wieder in industrienahen Projekten überprüft, wobei bisher auf Kooperationen mit der Automobilindustrie und ihren Zulieferern, der Bahnindustrie und aus der Robotik verwiesen werden kann.



Schloss Dagstuhl wurde 1760 von dem damals regierenden Fürsten Graf Anton von Öttingen-Soetern-Hohenbaldern erbaut. 1989 erwarb das Saarland das Schloss zur Errichtung des Internationalen Begegnungs- und Forschungszentrums für Informatik. Das erste Seminar fand im August 1990 statt. Jährlich kommen ca. 2600 Wissenschaftler aus aller Welt zu 40-45 Seminaren und viele sonstigen Veranstaltungen.



Die fortiss GmbH ist ein Innovationszentrum für softwareintensive Systeme in Form eines An-Instituts der Technischen Universität München. Als Forschungs- und Transferinstitut liegt der Fokus auf der angewandten Forschung zukünftiger Software- und Systemlösungen mit Schwerpunkt eingebettete und verteilte Systeme sowie Informationssysteme. Bearbeitete Themenfelder sind dabei unter anderem Modellierungstheorien einschließlich funktionaler, zeitlicher und nicht-funktionaler Aspekte, Werkzeugunterstützung zur Erstellung von domänenspezifischen, modellbasierten Entwicklungswerkzeugen, Architekturen insbesondere für langlebige und sicherheitskritische Systeme, sowie modellbasierte Anforderungsanalyse und Qualitätssicherung. Lösungen werden dabei vorwiegend in den Anwendungsfeldern Automobilindustrie, Luft- und Raumfahrt, Automatisierungstechnik, Medizintechnik, Kommunikationstechnik, öffentliche Verwaltung und Gesundheitswirtschaft erarbeitet.



Die Validas AG ist ein Beratungsunternehmen im Bereich Software-Engineering für eingebettete Systeme. Die Validas AG bietet Unterstützung in allen Entwicklungsphasen, vom Requirements- Engineering bis zum Abnahmetest. Die Auswahl und Einführung qualitätssteigernder Maßnahmen folgt dabei den Leitlinien modellbasierter Entwicklung, durchgängiger Automatisierung und wissenschaftlicher Grundlagen.



Besonderer Fokus der Arbeit des Fachgebiets "Systemanalyse und Modellierung" des Hasso Plattner Instituts liegt im Bereich der modellgetriebenen Softwareentwicklung für software- intensive Systeme. Dies umfasst die UML- basierte Spezifikation von flexiblen Systemen mit Mustern und Komponenten, Ansätze zur formalen Verifikation dieser Modelle und Ansätze zur Synthese von Modellen. Darüber hinaus werden Transformationen von Modellen, Konzepte zur Codegenerierung für Struktur und Verhalten für Modelle und allgemein die Problematik der Integration von Modellen bei der modellgetriebenen Softwareentwicklung betrachtet.

Dagstuhl-Workshop MBEEES:
Modellbasierte Entwicklung eingebetteter Systeme X
(Model-Based Development of Embedded Systems)

2005 startete die Dagstuhl-Workshopreihe „Modellbasierte Entwicklung eingebetteter Systeme“ mit der Feststellung, das zwar „(m)odellbasierte Vorgehensweisen mit der verstärkten Trennung von anwendungsspezifischen und implementierungsspezifischen Modellen (...) in der Softwareentwicklung in den letzten Jahren zunehmend an Bedeutung gewonnen“ haben, aber „(d)abei (...) die Bedeutung anwendungsorientierter Modelle mit domänenspezifischen Konzepten für diese Ansätze oft unterschätzt“ wird. Die Betonung dieses Aspekt war und ist daher ein wesentliches Anliegen über die zehn Ausgaben des Workshops hinweg, denn noch immer ist der – für eingebettete Systeme besonders wesentliche – Bereich der Regelungs- und Steuerungstechnik das primäre Ziel der industriellen modellbasierten Entwicklung.

Die beim ersten Workshop gefällte Prognose, „dass insbesondere durch Verbesserung der Entwicklungswerkzeuge, vor allem hinsichtlich Implementierungsqualität, Bedienkomfort und Analysefähigkeit, das Paradigma der modellorientierten Softwareentwicklung auch im Bereich eingebetteter Systeme an Einfluss gewinnen wird“, kann mit Sicherheit in einigen Industrien inzwischen als bestätigt angesehen werden. Trotz verbesserter Analyse- und Synthesefähigkeiten der Werkzeuge sind beispielsweise die in der Forschung aufgezeigten Potenziale bei der Unterstützung der Verifikationsaufgaben oder bei der Exploration des Entwurfsraums bei weitem noch nicht ausgeschöpft.

Das es sich bei diesen Industriezweigen – neben der Aeronautik vor allem die Automobilindustrie – insbesondere um solche handelt, bei denen die ingenieurmäßige Softwareentwicklung an den Produktgesamtkosten einen wesentlichen Anteil ausmachen, dieser jedoch durch Skalierungseffekte wie hohe Stückzahlen mit niedrigen Gesamtkosten oder mittlere Stückzahlen mit hohen Gesamtkosten, ist keine Überraschung. In diesen Zweigen – mit partiellen Durchdringungen der modellbasierten Entwicklung bis zu 90% der SW-Entwicklung – sind die substantiellen Kosten des notwendigen Kompetenzaufbaus einfacher umlegbar, sowie die Notwendigkeit der Investitionen am weitesten in das Bewusstsein der Führungsebene vorgebracht.

Damit sind die in allen Workshopausgaben genannten Ziele

- Austausch über Probleme und existierende Ansätze zwischen den unterschiedlichen Disziplinen (insbesondere Elektro- und Informationstechnik, Maschinenwesen/Mechatronik und Informatik)
- Austausch über relevante Probleme in der Anwendung/Industrie und existierende Ansätze in der Forschung

- Verbindung zu nationalen und internationalen Aktivitäten (z.B. Initiative des IEEE zum Thema Model-Based Systems Engineering, GI-AK Modellbasierte Entwicklung eingebetteter Systeme, GI-FG Echtzeitprogrammierung, MDA Initiative der OMG)

so aktuell wie in der 2005 abgehaltenen Ausgabe von MBEES.

Dass einige Themen in der modellbasierten Entwicklung auch wissenschaftlich noch intensiver Bearbeitung benötigten, zeigen die „Top Ten“ der wiederholten Nennungen adressierter Themen über die letzten neun Veranstaltungen hinweg:

- 9 Modellbasierte Validierung und Verifikation
- 8 Domänenspezifische Ansätze zur Modellierung von Systemen
- 8 Modellierung spezifischer Eigenschaften (z.B. Eigenschafteneigenschaften, Robustheit/Zuverlässigkeit, Ressourcenmodellierung)
- 6 Modellevolution
- 4 Syntheseverfahren und konstruktiver Einsatz von Modellen
- 4 Modelle in der architekturzentrierten Entwicklung
- 3 Bewertung der Qualität von Modellen
- 3 Durchgängigkeit und Integration von Modellen für eingebettete Systeme
- 2 Modellgestützte Design-Space Exploration
- 2 Formale Ansätze in der Modellbasierten Entwicklung

Konsequenterweise werden auch dies Jahr einige der Themen wieder aufgegriffen. Um so erfreulicher ist es, dass die Themenauswahl im 10. Jahr von MBEES insbesondere Themen aus der Erfahrung der Anwendung modellbasierter Anwendung in industrieller Praxis und akademischer Ausbildung einen breiten Platz gefunden haben.

Vor dem Hintergrund der zunehmenden Bedeutung von cyber-physischen Systemen und deren komplexen domänen- und disziplinübergreifenden Prozessen glauben wir, dass die Themen und Ziele der MBEES-Workshopreihe mehr denn je von Bedeutung sind.

Die Durchführung eines erfolgreichen Workshops ist ohne vielfache Unterstützung nicht möglich. Wir danken daher den Mitarbeitern von Schloss Dagstuhl.

Schloss Dagstuhl im März 2014,

Das Organisationskomitee:

Holger Giese, Hasso-Plattner-Institut an der Universität Potsdam

Michaela Huhn, TU Clausthal

Jan Philipps, Validas AG

Bernhard Schätz, fortiss GmbH

Mit Unterstützung von

Sergej Zverlov, fortiss GmbH

MDA für Altsysteme - Ein langer Weg

Michael Erskine, Alexander Schäffer, Rico Lieback
MBDA Deutschland GmbH, Schrobenhausen

Abstract: Dieser technische Beitrag beschreibt, wie ein Altsystem aus den 1990er Jahren erfolgreich umgestellt wurde: von der Methode Strukturierte Analyse mit dem Werkzeug *Teamwork* über ein Reverse Engineering-Modell mit UML bis hin zur vollstufigen MDA mit CIM, PIM, PSM, Modell-zu-Modell-Transformationen und Codegenerierung im Hybridbetrieb mit dem Altsystem.

1 Das Projekt

Bei dem Altprojekt handelt es sich um die eingebettete Echtzeit-Steuerungssoftware einer Waffenanlage für Lenkflugkörper am Hubschrauber. Die Entwicklung erfolgte in den 1990er Jahren.

Bereits damals wurde modellbasiert entwickelt. Als Methode diente *Strukturierte Analyse* nach DeMarco [DeM78] mit den Echtzeiterweiterungen *SA/RT* nach Hatley/Pirbhai [HP87] und Ward/Mellor [WM86]. Als Werkzeug diente *Teamwork*, das früher weit verbreitet war. Bei *SA/RT* wird ein Prozeß auf oberster Ebene so lange in Sub-Prozesse aufgeteilt, bis man eine Aufgabe hat, die von einem Software-Modul erfüllt werden kann. Kontroll- und Datenflüsse werden auf ähnliche Weise zerlegt und als In- und Output den Prozessen zugeordnet. *Teamwork* prüft, ob die Zerlegung syntaktisch korrekt und vollständig ist. Das *SA/RT*-Modell deckte die V-Modell-Phasen System-Analyse, System-Entwurf und Software-Analyse ab. Im nächsten Schritt wird die Software-Architektur mit der Methode *Strukturiertes Design* nach [YC79] modelliert (V-Modell-Phase Grobentwurf).

Die Implementierung erfolgte in Pascal. Die Vorgabe für die Implementierung bestand aus Pseudocode, Zustandsautomaten und Entscheidungstabellen auf der untersten Ebene des *SA/RT*-Modells. Eine große Anzahl selbstgeschriebener Hilfssoftware unterstützte die Implementierung und Dokumentation. Die Software umfaßt 1000 Dateien.

2 Motivation für die Umstellung

Als 2005 die Entwicklung beendet war und die nächste Phase – 20-30 Jahre lang Software-Pflege und -Änderung – begann, wurden mehrere kritische Aspekte identifiziert:

Compiler Der Pascal-Compiler ist obsolet. Er läuft auf einer obsoleten Hardware (VAX) mit obsoletem Betriebssystem (Ultrix). Zunächst wurde probeweise der Pascal-Code

automatisch nach C konvertiert. Für C existierte jedoch kein *zertifizierter* Cross-Compiler, was zu Problemen bei der Flugfreigabe geführt hätte. Umgesetzt wurde schließlich folgende Lösung: Die Hardware wurde durch einen virtuellen VAX-Rechner unter Windows ersetzt, auf dem der ursprüngliche Pascal-Compiler bis heute problemlos läuft – deutlich schneller als auf dem Original.

Werkzeug Teamwork Die Modellierungssoftware ist obsolet, d.h. man erhält keine Aktualisierungen mehr und kann keine neuen Lizenzen kaufen. Zum Zeitpunkt der Untersuchung (vor zehn Jahren) existierten noch alternative Werkzeuge, doch es war absehbar, daß auch sie bald obsolet werden. Als Lösung wurde versucht, SA/RT mit UML nachzubilden (siehe Kapitel 3), um mit einem aktuellen UML-Werkzeug weiterzuarbeiten.

Methode SA/RT Die Methode ist veraltet. Sie wird kaum noch gelehrt. Als Stand der Technik haben sich OOA und OOD mit UML durchgesetzt. Der Schritt, die Software im Rahmen eines Reverse Engineerings neu mit UML zu modellieren, schien zunächst zu groß, da zum Zeitpunkt der Entscheidung nur wenige Mitarbeiter ausreichende Erfahrung mit UML besaßen. Heute gilt das nicht mehr. Deshalb ist diese Lösung doch noch umgesetzt worden (siehe Kapitel 4).

Wissensverlust Die ursprünglichen Entwickler sind entweder aus der Firma ausgeschieden oder in neuen Projekten gebunden. Die Dokumentation der Software mit dem SA/RT-Modell erlaubt zwar einen detaillierten Zugang zur Software-Architektur, aber sie vermittelt nicht die Zusammenhänge und das Hintergrundwissen aus Sicht des Systems. Heute wird das Wissen Schritt für Schritt rekonstruiert und strukturiert jeweils in einem System- und Software-Modell gesichert.

3 Erster Lösungsversuch: SA/RT mit UML

Die höchste Priorität hatte die Obsoleszenz des Werkzeugs *Teamwork*. Ein Lösungsansatz bestand darin, die Methode SA/RT mit UML nachzubilden, um anschließend das Modell mit einem aktuellen UML-Werkzeug zu pflegen. Dabei wird die natürlichsprachlich beschriebene Methode der SA/RT-Modellierung in ein UML-Metamodell und ein UML-Profil für SA/RT überführt. Tatsächlich erweist sich die UML als so universell, daß es möglich ist, jedes Konstrukt der SA/RT in UML abzubilden (siehe [Vis05]).

Die Lösung wurde jedoch nicht umgesetzt. Sie hätte zwar die Werkzeug-Obsoleszenz beseitigt. Die Methode SA/RT wäre immer noch die gleiche: „Alter Wein in neuen Schläuchen“. Der Zugang, um Wissen über System und Software aufzubauen, wäre noch weiter erschwert worden, da einige Abbildungen sehr konstruiert wirken. Abbildung 1 zeigt dies am Beispiel einer Prozeßaktivierungstabelle. In der Tabellenform von SA/RT (oben) lassen sich viele Bedingungen miteinander verknüpfen. Die gleichen Bedingungen werden im UML-Aktivitätsdiagramm unübersichtlich (unten).

2-s1;1
radar_alarm_activate

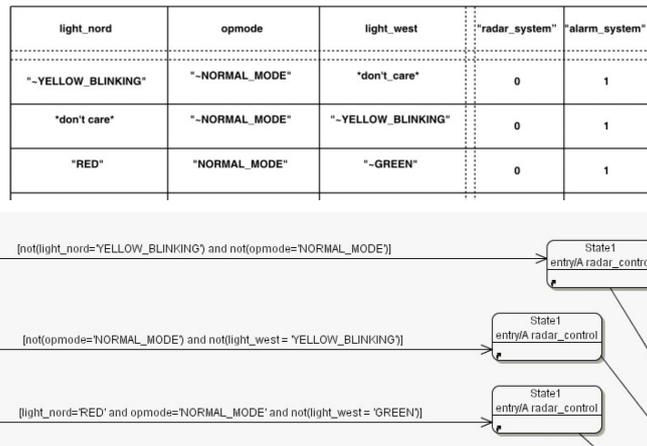


Abbildung 1: Oben: Prozeßaktivierungstabelle in SA/RT (Ausschnitt); unten: Umsetzung in UML als Aktivitätsdiagramm (Ausschnitt)

4 Reverse Engineering der Software-Architektur mit UML

Beim Ansatz im vorherigen Kapitel werden die Informationen aus dem alten Modell identisch in ein neues Modell überführt. Dabei bleibt der ursprüngliche Informationsgehalt unverändert. Was im alten Modell nicht enthalten war, fehlt auch im neuen.

Beim Reverse Engineering dagegen untersucht man die fertige Software und leitet daraus eine möglichst exakte Beschreibung ab. Das bietet die Möglichkeit, Informationen, die im alten System auf verschiedene Quellen verstreut waren (Modell, Datenbank, Dokumente) oder nicht modelliert waren, nach neuen Methoden in einer einzigen Quelle, dem neuen Modell, abzulegen.

Ein ähnlicher Weg in einem vergleichbaren Umfeld wird in [WS13] beschrieben. Auch dort lagen SA/RT-Modelle in Teamwork vor. Sie waren die Grundlage für Software-Anforderungsdokumente. In einem Reverse Engineering-Schritt wird ein UML-Modell erstellt, aus dem neue Anforderungsdokumente für die Wartungsphase generiert werden.

Auch im hier beschriebenen SA/RT-Modell sind Software-Anforderungen enthalten. Zusätzlich enthält es Informationen über die Software-Architektur, über Variablen und Datentypen sowie detaillierte Implementierungsvorgaben. Der Schwerpunkt des Reverse Engineering liegt auf Architektur und Feinentwurf, die in einem UML-Modell nach OOD-Prinzipien neu modelliert werden. Das Modell wurde 2013 umgestellt.

Datentypen Für jede Implementierungsdatei existiert eine zugehörige Deklarationsdatei, die alle verwendeten Datentypen und Variablen enthält. Die Deklarationsdatei wird im al-

ten System aus einer selbstgeschriebenen Datenbank heraus erzeugt. Beim Reverse Engineering wurde die Datenbank mit einem selbsterstellten Java-Hilfsprogramm analysiert, das einmalig die gewonnenen Informationen ins UML-Modell überträgt.

Die Pascal-Datentypen sind in einem eigenen Paket modelliert. Als Grundlage dienen die primitiven Typen Boolean, Char, Integer und Real. Auf sie stützen sich Konstanten, Arrays, Aufzählungstypen, Records und Mengen. Sie ließen sich einfach aus den existierenden Deklarationsdateien erzeugen. Kniffliger waren abgeleitete Typen, da die originale Deklaration nicht den Vater-Typ nennt, im UML-Modell aber diese Beziehung sichtbar gemacht werden sollte. Hier mußte zusätzliches Wissen von Hand ergänzt werden.

Im fertigen Modell ist es möglich, alle Deklarationsdateien aus dem Modell heraus zu erzeugen. Auf diese Weise wurde die proprietäre Datenbank des Altsystems abgelöst.

UML-Profil In einem eigenen UML-Profil sind Stereotypen definiert, die den Transfer der alten Bezeichner in die neue UML-Welt erleichtern. Stereotypen existieren zum Beispiel für spezielle Datentypen, für V-Modell-Begriffe wie SW-Komponente und SW-Modul, für Abkürzungen und Begriffsdefinitionen.

Komponenten und Module Die statische Architektur wird in einer Pakethierarchie abgebildet, wobei die Ebenen zusätzlich durch Stereotypen gekennzeichnet sind. Die Zugehörigkeit von Modulen zu einer Task wird durch Beziehungen modelliert. Die Hierarchie konnte beim Reverse Engineering aus der Verzeichnisstruktur abgelesen werden. Die Taskzugehörigkeit wurde dem Grobentwurf entnommen.

Die unterste Ebene stellen die SW-Module dar. Um den Bezug zum Modell nicht zu verlieren, sind die SA/RT-Diagramme zunächst als PDF-Dateien hinterlegt. Erst wenn ein Modul im Zuge der Wartung geändert werden muß, wird die PDF-Datei durch manuelle Modellierung von Sequenz-, Zustands- und Aktivitätsdiagrammen ersetzt. Es wird erwartet, daß nur ein kleiner Teil des Modells von Änderungen betroffen ist, etwa 5-10 Prozent.

Schnittstellen Die früheren Daten- und Kontrollflüsse sowie die Pascal-Prozeduren werden nun als Ports modelliert, sowohl intern zwischen den Modulen als auch extern. Sie enthalten die Information darüber, wer für Variablen verantwortlich ist, wer sie liest oder schreibt und wer eine Prozedur aufruft. Diese Information war früher nur aufwendig über eine Datenbank zu ermitteln.

Die Datenbank diente als primäre Quelle für das Reverse Engineering. Ein Modul, das eine Variable schreibt, ist im UML-Modell verantwortlich für die Variable – sie wird zum Attribut der entsprechenden UML-Klasse. Falls mehrere Module schreibend zugreifen, wird die Variable dennoch nur einem Modul als Attribut zugeordnet; die anderen Module greifen schreibend über die Schnittstelle auf dieses Attribut zu.

Das Beispiel in Abbildung 2 zeigt für das SW-Modul *OS_Time_Counter_Int_Time*, daß es die Variable *azeit* vom SW-Modul *BS_Do_WA_Bus_Transfer* liest, die Variable *duration_for_integration_time_exceeded* schreibt und dem SW-Modul *OS_Int_Time_Trans*

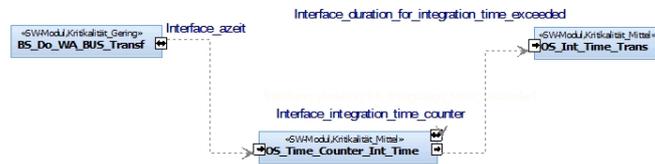


Abbildung 2: Klassendiagramm, das die Verwendung der Variablen zeigt

zur Verfügung stellt; die Variable *integration_time_counter* wird nur innerhalb des SW-Moduls gelesen und geschrieben.

Diagramme Einige Informationen werden zusätzlich als Diagramme dargestellt, um den Zugang zum Modell zu erleichtern. Das gilt zum Beispiel für die Zugehörigkeit von Modulen zu Komponenten und zu den Tasks sowie für die Nutzung von Variablen an den Ports innerhalb einer Komponente. Die Diagramme konnten größtenteils automatisch durch ein selbstgeschriebenes Java-Hilfsprogramm erzeugt werden.

Modellierungswerkzeug Um zukünftigen Obsoleszenzen vorzubeugen, wurde darauf geachtet, so wenige werkzeugspezifische Konstrukte wie möglich zu verwenden. Als Werkzeug wird *Rational Rhapsody* eingesetzt. Eigene Java-Programme greifen über die API direkt auf das Modell zu. Um noch unabhängiger vom Hersteller zu werden, wurde versucht, die XMI-Repräsentation des Modells zu bearbeiten. Das stellte sich als unpraktikabel heraus. Die XMI-Konstrukte sind nicht ausreichend beschrieben, und die Bearbeitungszeit ist um ein Vielfaches schlechter als beim Zugriff über die API.

Bewertung Das UML-Modell konnte einschließlich Diagramme größtenteils automatisch erstellt werden. Das UML-Modell bildet die Software-Architektur sehr gut ab und erlaubt eine bessere Analyse der Software als das SA/RT-Modell. Durch den Einsatz von Regeln, die von einem Skript im Modell automatisch überprüft werden, wird die Einhaltung der Modellierungsrichtlinie und die Konsistenz und Vollständigkeit des Modells geprüft.

Neue Analyse-Hilfsmittel sind nun leichter zu realisieren. Die alte SA/RT-Entwicklungsumgebung bestand aus einer Vielzahl eigenentwickelter Hilfsmittel wie C-Programme, Shell-Skripte und Datenbanken, die *ad hoc* für einen bestimmten Zweck erstellt wurden. In der neuen Entwicklungsumgebung können alle Informationen aus einer Quelle, aus dem UML-Modell, gewonnen werden.

Zeitgleich wird ein SysML-Modell erstellt, das das Systemwissen nachmodelliert. Die beiden Modelle können zukünftig miteinander verknüpft werden, durch Beziehungen zwischen den Modellen oder durch ein Anforderungsverfolgungswerkzeug.

Der Weg stellt eine Lösung für die ursprünglichen Obsoleszenzen dar: das Werkzeug wird durch ein Standard-UML-Werkzeug abgelöst, die Methode orientiert sich an Objektorien-

tiertem Design und das Wissen, das von neuen Mitarbeitern aufgebaut wird, findet vordefinierte Ablageorte sowohl im System-, als auch im Software-Modell. Damit befindet sich das Projekt wieder auf dem Stand der Technik.

5 Model Driven Architecture im Hybridbetrieb mit dem Altsystem

In der Lenkflugkörper-Industrie wird seit langem modellbasiert entwickelt. Aktuell stützen sich die meisten Projekte auf die Möglichkeiten, die Werkzeuge wie *Rose RT* oder *Rhapsody* anbieten. Dabei muß die vom Werkzeughersteller vorgegebene Entwicklungsstrategie eingehalten werden, um den eingebauten Codegenerator innerhalb eines Frameworks des Werkzeugherstellers nutzen zu können. Ein schnelles Ergebnis wird mit der Abhängigkeit von einem Werkzeug erkauft. Das führt später zu Problemen, falls das Werkzeug gewechselt werden muß.

In einem Pilotprojekt hat die Firma gute Erfahrungen mit MDA gesammelt. Dabei handelte es sich um eine Neuentwicklung. Anstatt auf das Werkzeug zu setzen, wurden konsequent alle Transformatoren und Generatoren als Java- oder VBA-Programm selbst implementiert. Das Ergebnis sah aus wie der von Hand implementierte Code eines ähnlichen Projekts, wurde aber komplett aus dem Modell generiert. Das führte zu der Frage, ob sich das MDA-Vorgehen auch in diesem Altsystem umsetzen läßt.

In der Pflege- und Änderungsphase ist ein typisches Szenario, daß nur eine Teilfunktionalität betrachtet wird. Im Rahmen einer Masterarbeit wurde ein Vorgehen entwickelt, wie neben dem UML-Modell ein neues MDA-Modell entstehen kann, das Funktion um Funktion die alte Software schrittweise ablösen kann (siehe [Lie13]).

Dieser Hybridansatz, Altsystem gekoppelt mit MDA-Modell, ist ein neuer Ansatz in unserer Domäne. Über ähnliche Ansätze aus anderen Industrien gibt es wenige Berichte. Aus der Eisenbahn-Industrie wird über ein Pilotprojekt berichtet, das den Einsatz von MDA für Altsysteme anhand einer ausgewählten Funktionalität untersucht [MRA05]; dabei bleibt offen, ob der aus dem Modell generierte Code in Betrieb genommen wurde. Unser Ansatz wird ab 2014 im operativen Code eingesetzt.

5.1 CIM und PIM

Um eine Teilfunktion zu ändern oder neu zu implementieren, muß sie verstanden werden. Dafür werden alle verfügbaren Quellen herangezogen: alte Spezifikationen, das alte SA/RT-Modell, die neu gestaltete UML-Software-Architektur und der Quellcode. Das Wissen wird im *Computational Independent Model (CIM)* strukturiert abgelegt: in Form von Anwendungsfällen, Fachklassen und Aktivitätsdiagrammen. Im CIM geht es primär darum, die fachliche Aufgabe zu modellieren, ohne an eine konkrete Lösung zu denken.

Die fachliche Lösung wird im *Platform Independent Model (PIM)* implementiert. Hier werden Ereignisse, die im CIM noch abstrakt waren, konkreten Methoden zugeordnet, Klassen

werden zu Komponenten zusammengefaßt, Schnittstellen und Abhängigkeiten modelliert. Alle Methoden und Attribute werden typisiert; dabei entstehen etliche zusätzliche Klassen. Die Methoden werden vollständig mit Logik gefüllt. Um den Aufwand für die folgenden Transformationen gering zu halten, werden zunächst nur Sequenzdiagramme verwendet. Sie stellen Konstrukte wie Schleifen und Entscheidungen zur Verfügung, lassen sich formal leicht einer Klasse zuordnen und können über die API einfach ausgelesen und analysiert werden. Mit Hilfe der *Object Constraint Language* werden zusätzliche Informationen modelliert (siehe Beispiel in Abbildung 3). Bei Bedarf werden später Aktivitäts- und Zustandsdiagramme unterstützt.

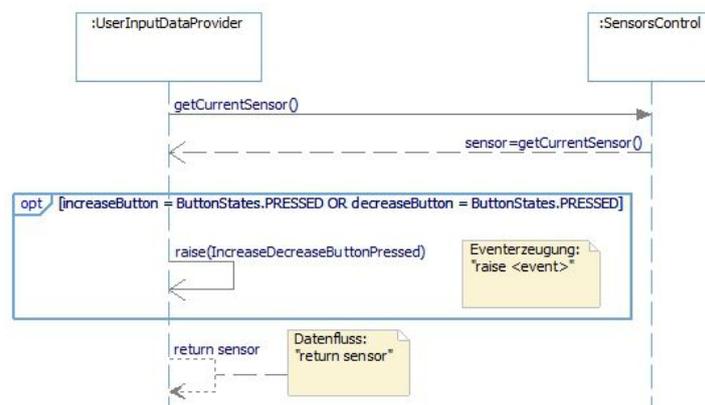


Abbildung 3: Methoden werden im PIM als Sequenzdiagramm modelliert

Das PIM enthält drei Schnittstellen zwischen Alt- und Neusystem (siehe Beispiel in Abbildung 4). Die Komponenten des Altsystems werden als Akteure betrachtet. In der ersten Schnittstelle sind die Daten enthalten, die das Altsystem zur Verfügung stellt, gekennzeichnet mit dem Stereotyp `«uses»`. Die zweite Schnittstelle enthält den Aufruf einer Methode aus dem Altsystem in das Neusystem (Stereotyp `«externalCall»`). In der dritten Schnittstelle sind die Daten enthalten, die vom Neusystem berechnet werden und dem Altsystem zur Weiterverarbeitung bereitgestellt werden (Stereotyp `«provides»`).

5.2 Das Plattformmodell

Das Plattformmodell (PM) enthält die Beschreibung der Zielplattform. Es bildet die Brücke zwischen PIM auf der einen Seite und konkreten Gegebenheiten andererseits:

Software-Architektur Im PM werden die Klassen des PIM neu miteinander verknüpft, um die nicht-funktionalen Qualitätsanforderungen zu erfüllen. Das geschieht durch Namenskonventionen und stereotypisierte Beziehungen. Im PM kommen auch neue Klassen hinzu, zum Beispiel um bestimmte Aufgaben an der Schnittstelle zu über-

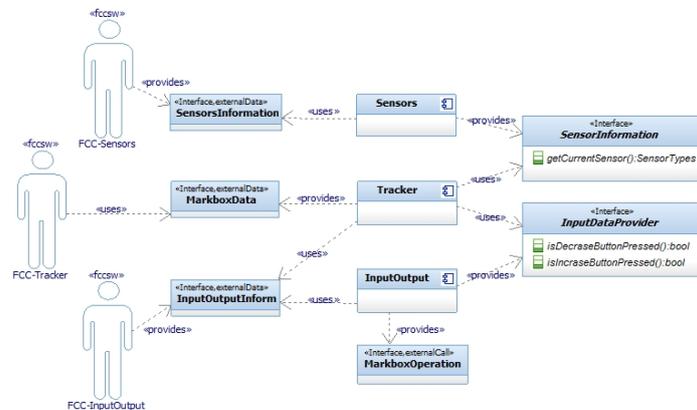


Abbildung 4: Schnittstelle zwischen Alt- und Neusystem

nehmen, die im PIM fehlen, da sie aus fachlicher Sicht nicht relevant sind.

Datentypen Das PIM arbeitet mit modellspezifischen Datentypen. Im PM werden sie auf Datentypen abgebildet, die vom Ziel-Compiler verarbeitet werden können. Auch hier kommen stereotypisierte Beziehungen zum Einsatz, um sowohl primitive als auch zusammengesetzte Typen auf konkrete Pascal-Datentypen und -Konzepte abzubilden.

Datenschnittstelle zwischen Altsystem und Neusystem Das PIM enthält zwar schon die Schnittstelle zwischen dem alten und dem neuen System. Aber erst im PM wird diese Schnittstelle konkret modelliert. Dabei werden die Variablen aus dem prozeduralen Altsystem auf Attribute der objektorientierten Modellierung im PIM abgebildet. Auf die gleiche Weise werden Datentypen aufeinander abgebildet.

Aufrufschnittstelle Der Zugriff auf Daten des Altsystems darf die Konsistenz der Daten im laufenden Betrieb nicht verletzen. Deshalb enthält das PM Informationen darüber, auf welche Daten vor der Ausführung einer Methode lesend zugegriffen wird und welche Daten nach der Ausführung geschrieben werden. Der Methodenaufruf im Neusystem selbst wird im PM als *Event* modelliert.

5.3 Das PSM als Zwischenstufe vor dem Code

Wollte man aus dem objektorientierten PIM und dem PM, das unter Qualitätsaspekten wie Wartbarkeit und Verständlichkeit modelliert wird, direkt Code erzeugen, wäre der Schritt zu prozeduralem Pascal sehr groß. Deshalb wird aus PIM und PM in einer Modell-zu-Modell-Transformation automatisch das *Platform Specific Model (PSM)* erzeugt. Im PSM sind nun die fachlichen Aspekte aus dem PIM und die technischen Aspekte aus dem PM

vereint in einem einzigen Modell. Von den Diagrammen werden nur die Sequenzdiagramme übernommen. Technische Aspekte werden optimiert und alle objektorientierten Konzepte eliminiert. Zum Beispiel:

Assoziationen werden in Attribute umgewandelt. *Vererbung* wird in die abgeleitete Klasse verlagert, da Pascal keine Vererbung kennt. *Events* werden Prozeduraufrufe. *Serviceorientierte Architektur* wird in eine Pakethierarchie umgewandelt. *Zugriffsschnittstellen* werden vereinfacht und dort aufgelöst, wo sie im PM nur existierten, um die Konsistenz sicherzustellen.

5.4 Codegenerierung

Das PSM enthält nur noch Konstrukte, die sich nahezu sofort auf Pascal-Konstrukte abbilden lassen. Deshalb ist die Erzeugung von Code in einer Modell-zu-Text-Transformation relativ einfach und schematisch umsetzbar. Da der Codegenerator aus eigener Hand ist, kann auf Besonderheiten des Compilers und des Projekts Rücksicht genommen werden. Aus Performancegründen werden beispielsweise im Code nur globale Variablen anstelle von Übergabe- und Rückgabeparametern verwendet; im MDA-Ansatz bleiben private Attribute bis zum letzten Schritt privat und werden erst bei der Codegenerierung in globale Variablen umgewandelt. Der generierte Code sieht genauso aus wie die von Hand implementierten Pascal-Dateien des Altsystems.

5.5 Bewertung

Die große Stärke des MDA-Vorgehens liegt in der *Separation of Concerns*. Jedes Teilmodell steuert einen anderen Aspekt bei. Im CIM läßt sich Wissen strukturiert ablegen. Im PIM kann man sich auf die Lösung der fachlichen Aufgabe der Software konzentrieren. Projektspezifische Aspekte wie die Software-Architektur, Programmiersprache und Lösungsmuster sind im PM gut aufgehoben. Bei der Transformation von PIM und PM ins PSM werden projektabhängige Aspekte weiter konkretisiert. Der Codegenerator schließlich kann sich rein auf die Optimierung des ablauffähigen Codes konzentrieren.

Diese Stärke fällt besonders dann auf, wenn man das alte Modell in SA/RT im Vergleich zum MDA-Modell sieht. Das alte Modell enthält alle Aspekte gleichzeitig, von Systemaspekten über Softwarearchitektur bis hinunter zu Pascal-spezifischen Konstrukten. Das SA/RT-Modell bewegt sich auf der Ebene eines PSM. Das erschwert den Zugang zum Wissen so sehr, daß das Modell praktisch seit Beginn der Software-Pflege- und Änderungsphase nicht mehr genutzt wird.

Beeindruckend ist beim MDA-Vorgehen der Übergang von objektorientierter Analyse und Design zu prozeduraler Programmierung. CIM und PIM sehen aus wie ein modernes Projekt. Jeder, der UML kennt, kann sich in das Modell einarbeiten. Der generierte Code fügt sich nahtlos in die prozedurale Implementierung des Altsystems ein.

Sowohl der Modell-Transformator als auch der Code-Generator sind eigenentwickelt. Sie funktionieren nur mit diesem Modell. Dadurch sind sie zwar nicht universell, aber dafür schlank und effizient. Sie erzeugen genau das, was im Projektumfeld erwartet wird. Das erhöht die Akzeptanz und das Vertrauen in das Vorgehen.

6 Resümee

Der Weg, ein Altsystem zu modernisieren, ist lang, da es während der Umstellung durchgehend einsatzbereit bleiben muß. Dabei helfen viele kleine Schritte. Der wichtigste Schritt in diesem Projekt ist die Ablösung von SA/RT mit Teamwork durch UML. Dadurch werden drängende Obsoleszenzen beseitigt. Gleichzeitig eröffnen sich viele neue Möglichkeiten, die in der alten Umgebung nur aufwendig realisierbar wären. Die interessanteste Möglichkeit ist der schrittweise Übergang zur Objektorientierung und Codegenerierung mit Hilfe der MDA.

Literatur

- [DeM78] Tom DeMarco. *Structured Analysis and System Specification*. Yourdon Press, 1978.
- [HP87] Derek J. Hatley und Imtiaz A. Pirbhai. *Strategies for Real-Time System Specification*. Dorset House, 1987.
- [Lie13] Rico Lieback. MDA für Altsysteme. Werkzeug und Prozess zur schrittweisen Umsetzung von bestehenden zuverlässigen Systemen nach MDA. Masterarbeit, Hochschule Augsburg, 2013.
- [MRA05] Anthony MacDonald, Danny Russell und Brenton Atchison. Model-Driven Development within a Legacy System: An Industry Experience Report. *Software Engineering Conference, Australian*, 0:14–22, 2005.
- [Vis05] Andrea Visnyovszki. Entwicklung eines Konzeptes zur Übertragung von SA/RT-Modellen nach UML (nach einer Spezifikation für Lenkflugkörper-Systeme). Diplomarbeit, Ludwig-Maximilians-Universität München, 2005.
- [WM86] Paul Ward und Steve Mellor. *Structured Development for Real-Time Systems*. Yourdon Press, 1986.
- [WS13] Thomas Weyrath und Herbert Schreyer. Saving the Software Specification by Converting the old SA/RT Models into UML. In *5th European Conference for Aeronautics and Space Sciences*, München, Deutschland, Juni 2013.
- [YC79] Edward Yourdon und Larry L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice Hall, 1979.

Towards Symbolic Causality Checking using SAT-Solving

Adrian Beer¹, Uwe Kühne², Florian Leitner-Fischer¹, Stefan Leue¹

¹University of Konstanz, ²EADS Deutschland GmbH

Abstract: With the increasing size and complexity of modern safety critical embedded systems, the need for automated analysis methods is growing as well. Causality Checking is an automated technique for formal causality analysis of system models. In this paper we report on work in progress towards an Symbolic Causality Checking approach. The proposed approach is based on bounded model checking using SAT solving which is known to be efficient for large and complex system models.

1 Introduction

The size and complexity of modern software-driven and safety critical systems is increasing at a high rate. In this situation, classical manual safety analysis techniques like reviewing, fault tree analysis [VGRH02] and failure mode and effect analysis [Int91] can only be applied to very limited parts of the architecture of a system. Furthermore, these techniques are more suitable for analyzing faults in hardware systems rather than in software driven embedded systems. The demand for automated methods and tools supporting the safety analysis of the architecture of software-driven safety-critical systems is growing.

In previous work, an algorithmic, automated safety-analysis technique called *causality checking* was proposed [LFL13a]. Causality checking is based on model checking. In model checking, the model of the system is given in a model checker specific input language. The property is typically given in some temporal logic. The model checker verifies whether the model acts within the given specifications by systematically generating the state space of the model. If the model does not fulfill the specification, an error trace leading from the initial state of the model to the property violation is generated. One trace only represents one execution of the system. In order to understand all possibilities of how an error can occur in a system, all possible error traces have to be generated and inspected. Manually locating reasons for property violations using these traces is problematic since they are often long, and typically large numbers of them can be generated in complex systems. Causality checking is an algorithmic, automated analysis technique working on system traces which supports explaining why a system violates a property. It uses an adaption of the notion of actual causality proposed by Halpern and Pearl [HP05]. The result of the causality checking algorithm is a combination of events that are causal for an error to happen. The event combinations are represented by formulae in Event Order Logic (EOL) [LFL13c], which can be fully translated into LTL, as is shown in [BLFL14]. The EOL formulae produced by causality checking represent the causal events in a more compact way than counterexamples since they only contain the events and the relation between those events that are considered to be causal for a property violation. It was shown

that the explicit-state causality checking approach is efficient for system models for which state-of-the-art explicit model checking is efficient as well [LFL13a].

Although the explicit-state causality checking method was shown to be efficient for small to medium sized models, for system models that cannot be efficiently processed by explicit state model checkers the causality computation is also not efficient. In this paper we propose a new causality checking approach based on Bounded Model Checking (BMC) [BCCZ99]. BMC can efficiently find errors in very large systems where explicit model checking runs out of resources. One drawback of BMC is that it is not a complete technique since it cannot prove the absence of errors in a system beyond a predefined bound on the length of the considered execution traces. For the proposed symbolic causality checking method this means that completeness for the computed causalities can only be guaranteed for system runs up to the given bound. In explicit causality checking all traces through a system have to be generated in order to gain insight into the causal events. The symbolic causality checking approach presented in this paper uses the underlying SAT-solver of the bounded model checker in order to generate the causal event combinations in an iterative manner. This means that only those error traces are generated that give new insight into the system. Traces that do not give new information are automatically excluded from the bounded model checking algorithm by constraining the SAT-solver with the already known information. With this technique a large number of error traces can be ruled out that would need to be considered in the explicit approach, which contributes to the efficiency of the symbolic approach. We implemented this approach as an addition to the NuSMV2 model checker [CCG⁺02].

In [LFL11a, LFL11b] we presented a tool based approach called QuantUM that allows for specification of dependability characteristics and requirements directly within a system or software architecture modeled in UML [uml10] or SysML [Sys10]. The system models are automatically translated into the input language of different model checkers, for instance the model checker SPIN [Hol03]. Afterwards, the integrated explicit causality checker calculates the causal events for a property violation and displays the results in terms of dynamic Fault Trees [VGRH02]. In [LFL13b] a combination of the explicit causality checking and the probability computation was shown where probabilities for the causality classes can be calculated. The probabilities can be tagged to the Fault Trees. The integration of causality checking into the QuantUM tool chain enables the applicability of causality computations in an model-based engineering environment. The integration of the symbolic causality checking presented in this paper can be done in a similar way.

The remainder of the paper is structured as follows. In Section 2 we will present the foundations of our work which includes bounded model checking and the notion of causality. Section 3 is devoted to the new symbolic approach to causality computation. In Section 4 we evaluate the symbolic approach in comparison to the explicit causality checking. Related work will be discussed in Section 5 before we conclude in Section 6.

2 Preliminaries

2.1 Running Example

We will illustrate the formal framework that we present in this paper using the running example of a simple railroad crossing system. In this system, a train can approach the

crossing (Ta), enter the crossing (Tc), and finally leave the crossing (Tl). Whenever a train is approaching, the gate shall close (Gc) and will open again when the train has left the crossing (Go). It might also be the case that the gate fails (Gf). The car approaches the crossing (Ca) and crosses the crossing if the gate is open (Cc) and finally leaves the crossing (Cl). We are interested in finding those events that are causal for the hazard that the car and the train are in the crossing at the same time.

2.2 System Model

The system model that we use in this paper is that of a transition system [BK08]:

Definition 1 (Transition System). *A transition system M is a tuple $(S, \mathcal{A}, \rightarrow, I, AP, L)$ where S is a finite set of states, \mathcal{A} is a finite set of actions/events, $\rightarrow \subseteq S \times \mathcal{A} \times S$ is a transition relation, $I \subseteq S$ is the set of initial states, AP is the set of atomic propositions, and $L: S \rightarrow 2^{AP}$ is a labeling function.*

Definition 2 (Execution Trace). *An execution trace π in M is defined as an alternating sequence of states $s \in S$ and actions $a \in \mathcal{A}$ ending with a state. $\pi = s_0 \alpha_1 s_1 \alpha_2 s_2 \dots \alpha_n s_n$, s.t. $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ for all $0 \leq i < n$.*

An execution sequence which ends in a property violation is called an error trace or a counterexample. In the railroad crossing example, $s_0 \xrightarrow{Ta} s_1 \xrightarrow{Gf} s_2 \xrightarrow{Tc} s_3 \xrightarrow{Ca} s_4 \xrightarrow{Cc} s_5$ is a counterexample, because the train and the car are inside the crossing at the same time.

2.3 Linear Temporal Logic

We use the standard syntax and semantics of the Linear Temporal Logic (LTL) as introduced by Pnueli [Pnu77]. The operators \bigcirc , \square , \diamond and \mathbf{U} are used to express temporal behavior, such as “in the next state sth. happens” (\bigcirc), “eventually sth. happens” (\diamond) or “sth. is always true” (\square). The \mathbf{U} -operator denotes the case that “ φ_1 has to be true until φ_2 holds”.

There are two non-disjoint classes of LTL properties, safety and liveness properties. Safety properties can be violated by a finite prefix of an infinite path, while liveness properties can only be violated by an infinite path. For now, causality checking has only been defined for safety properties.

The property that we want to express in the railroad crossing is that the train and the car shall never be in the crossing at the same time: $\square \neg (Tc \wedge Cc)$.

2.4 Event Order Logic

The Event Order Logic [LFL13c] (EOL) can be fully translated into LTL as was shown in [BLFL14]. EOL captures the occurrence and order of events on a trace through a transition system.

Definition 3 (Syntax of the Event Order Logic). *Simple event order logic formulae are defined over the set A of event variables:*

$$\phi ::= a \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg \phi$$

where $a \in A$ and ϕ, ϕ_1 and ϕ_2 are simple EOL formulae. Complex EOL formulae are formed according to the following grammar:

$$\psi ::= \phi \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2 \mid \psi_1 \wedge \psi_2 \mid \psi_1 \wedge [\phi \mid \psi_1 \wedge] \phi \mid \psi_1 \wedge < \phi \wedge > \psi_2$$

where ϕ is a simple EOL formula and ψ, ψ_1 and ψ_2 are complex EOL formulae.

We define that a transition system M satisfies the EOL formula ψ , written as $M \models_e \psi$ iff $\exists \pi \in M. \pi \models_e \psi$. The informal semantics of the operators can be given as follows.

- $\psi_1 \wedge \psi_2$: ψ_1 has to happen before ψ_2 .
- $\psi_1 \wedge [\phi$: ψ_1 has to happen at some point and afterwards ϕ holds forever.
- $\phi \wedge] \psi_1$: ϕ has to hold until ψ_1 holds.
- $\psi_1 \wedge < \phi \wedge > \psi_2$: ψ_1 has to happen before ψ_2 , and between ψ_1 and ψ_2 , ϕ has to hold all the time.

For example, the formula $Gc \wedge Tc$ states that the gate has to close before the train enters the crossing. The full formal semantics definition for EOL is given in [LFL13c].

2.5 Causality Reasoning

Our goal is to identify the events that cause a system to reach a property violating state. Therefore, it is necessary to formally define what “cause” means in our context. We will use the same definition of causality that was proposed in [KLFL11] as an extension of the *structural equation model* by Halpern and Pearl [HP05]. In particular this extension accounts for considering the order of events in a trace to be causal. For example, an event a may always occur before an event b for an error to happen, but if b occurs first and a afterwards the error disappears. In this case, a occurring before b is considered to be causal for the error to happen.

Definition 4 (Cause for a property violation [HP05, LFL13a]). *Let π, π' and π'' be paths in a transition system M . The set of event variables is partitioned into sets Z and W . The variables in Z are involved in the causal process for a property violation while the variables in W are not. The valuations of the variables along a path π are represented by $val_z(\pi)$ and $val_w(\pi)$, respectively. ψ_\wedge denotes the rewriting of an EOL formula ψ where the ordering operator \wedge is replaced by the normal EOL operator \wedge , all other EOL operators are left unchanged. An EOL formula ψ consisting of event variables $X \subseteq Z$ is considered to be a cause for an effect represented by the violation of an LTL property φ , if the following conditions hold:*

- **AC 1:** *There exists an execution π for which both $\pi \models_e \psi$ and $\pi \not\models_l \varphi$*
- **AC 2.1:** *$\exists \pi'$ s.t. $\pi' \not\models_e \psi \wedge (val_x(\pi) \neq val_x(\pi') \vee val_w(\pi) \neq val_w(\pi'))$ and $\pi' \models_l \varphi$. In other words, there exists an execution π' where the order and occurrence of events is different from execution π and φ is not violated on π' .*
- **AC 2.2:** *$\forall \pi''$ with $\pi'' \models_e \psi \wedge (val_x(\pi) = val_x(\pi'') \wedge val_w(\pi) \neq val_w(\pi''))$ it holds that $\pi'' \not\models_l \varphi$ for all subsets of W . In words, for all executions where the events in X have the value defined by $val_x(\pi)$ and the order defined by ψ , the value and order of an arbitrary subset of events on W has no effect on the violation of φ .*
- **AC 3:** *The set of variables $X \subseteq Z$ is minimal: no subset of X satisfies conditions AC 1 and AC 2.*

- *OC 1: The order of events represented by the EOL formula ψ is not causal if the following holds: $\pi \models_e \psi$ and $\pi' \not\models_e \psi$ and $\pi' \not\models_e \psi_\wedge$*

The EOL formula $Gf \wedge ((Ta \wedge (Ca \wedge Cc)) \wedge_{<} \neg Cl \wedge_{>} Tc)$ is a cause for the occurrence of the hazard in the railroad crossing example since it fulfills all of the above defined conditions (AC 1-3, OC 1).

2.6 Bounded Model Checking

The basic idea of Bounded Model Checking (BMC) [BCCZ99] is to find error traces, also called counterexamples, in executions of a given system model where the length of the traces that are analyzed are bounded by some integer k . If no counterexample is found for some bound k , it is increased until either a counterexample is found or some pre-defined upper bound is reached. The BMC problem is efficiently reduced to a propositional satisfiability problem, and can be solved using propositional SAT solvers. Modern SAT solvers can handle satisfiability problems in the order of 10^6 variables.

Given a transition system M , an LTL formula f and a bound k , the propositional formula of the system is represented by $[[M, f]]_k$. Let s_0, \dots, s_k be a finite sequence of states on a path π . Each s_i represents a state at time step i and consists of an assignment of truth values to the set of state variables. The formula $[[M, f]]_k$ encodes a constraint on s_0, \dots, s_k such that $[[M, f]]_k$ is satisfiable iff π is a witness for f . The propositional formula $[[M, f]]_k$ is generated by unrolling the transition relation of the original model M and integrate the LTL property in every step s_i of the unrolling. The generated formula $[[M, f]]_k$ of the whole system is passed into a propositional SAT solver. The SAT solver tries to solve $[[M, f]]_k$. If a solution exists, this solution is considered to be a witness to the encoded LTL property.

3 Symbolic Causality Checking

3.1 Event Order Normal Form

In order to efficiently store the event orderings and occurrences in the symbolic causality algorithm it is necessary to use a normal form. This normal form is called event order normal form (EONF) [BLFL14]. EONF permits the unordered *and*- (\wedge) and *or*-operator (\vee) only to appear in a formula if they are not sub formulas in any ordered operator and only *and*-operators (\wedge) if they are sub formulas of the between operators $\wedge_{<}$ and $\wedge_{>}$. For instance, the EOL formula $Ta \wedge Gc \wedge Tc$ can be expressed in EONF as $\psi_{\text{EONF}} = (Ta \wedge Gc) \wedge (Gc \wedge Tc) \wedge (Ta \wedge Tc)$.

3.2 EOL Matrix

For the symbolic causality computation with bound k we focus on sequence of events $\pi_e = e_1 e_2 e_3 \dots e_k$ derived from paths of type $\pi = s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} s_2 \dots$. We use a matrix in order to represent the ordering and occurrence of events along a trace. This matrix is called EOL matrix.

Definition 5 (EOL matrix). *Let $E = \{e_1, e_2, e_3, \dots, e_k\}$ an event set and $\pi_e = e_1 e_2 e_3 \dots e_k$ the corresponding sequence. The function o is defined for entries where $i \neq j$ and the func-*

tion d is defined for entries where $i = j$:

$$o(e_i, e_j) = \begin{cases} \{TRUE\} & \text{if } e_i \wedge e_j \\ \phi & \text{if } e_i \wedge \phi \wedge e_j \\ \emptyset & \text{otherwise} \end{cases} \quad d(e_i) = \begin{cases} \phi & \text{if } \phi \wedge e_i \\ \emptyset & \text{otherwise} \end{cases}$$

The EOL matrix M_E is created as follows:

$$M_E = \begin{pmatrix} d(e_1) & o(e_1, e_2) & \cdots & o(e_1, e_k) \\ o(e_2, e_1) & d(e_2) & \cdots & o(e_2, e_k) \\ \vdots & \vdots & \ddots & \vdots \\ o(e_k, e_1) & o(e_k, e_2) & \cdots & d(e_k) \end{pmatrix}$$

where the generated entries in the matrix are sets of events or the constant set $\{TRUE\}$. The empty set \emptyset indicates that no relation for the corresponding event configuration was found.

The special case $e \wedge \phi$ is not considered in function d because this will never occur when analyzing safety properties.

Definition 6 (Union of EOL Matrices). *Let M_E, M_{E_1}, M_{E_2} be EOL Matrices with the same dimensions. The EOL matrix M_E is the union of M_{E_1} and M_{E_2} according to the following rule:*

$$M_{E(i,j)} = M_{E_1(i,j)} \cup M_{E_2(i,j)} \quad (1)$$

for every entry (i, j) in the matrices.

The union of two EOL matrices represents the component-wise disjunction of two matrices. The EOL matrix M_E for an example event sequence in the railroad crossing $\pi = \text{Ca Cc Gf}$ and a refinement EOL Matrix $M'_E = M_E \cup M_{E_{\pi'}}$ using the sequence $\pi' = \text{Gf Ca Cc}$ is created as follows:

$$\begin{array}{l} e_1 = \text{Ca} \\ e_2 = \text{Cc} \\ e_3 = \text{Gf} \end{array} \quad M_E = \begin{pmatrix} \emptyset & \{TRUE\} & \{TRUE\} \\ \emptyset & \emptyset & \{TRUE\} \\ \emptyset & \emptyset & \emptyset \end{pmatrix} \quad M'_E = \begin{pmatrix} \emptyset & \{TRUE\} & \{TRUE\} \\ \emptyset & \emptyset & \{TRUE\} \\ \{TRUE\} & \{TRUE\} & \emptyset \end{pmatrix} \quad (2)$$

The information stored in a EOL matrix can be translated back into an EOL formula in EONF. As was shown in [BLFL14] every EOL formula can be translated into an equivalent LTL formula. This translated LTL formula is then translated further into propositional logic [BCCZ99].

3.3 The Algorithm

In Figure 1 the informal iteration schema of the proposed algorithm is presented. The inputs to the algorithm are the model M , the property ϕ to check and an upper bound k_{max} for the maximum length of individual counterexamples (CX).

1. the algorithm starts at level $k = 0$.
2. If no CX is found the bound is increased until the next CX is found.

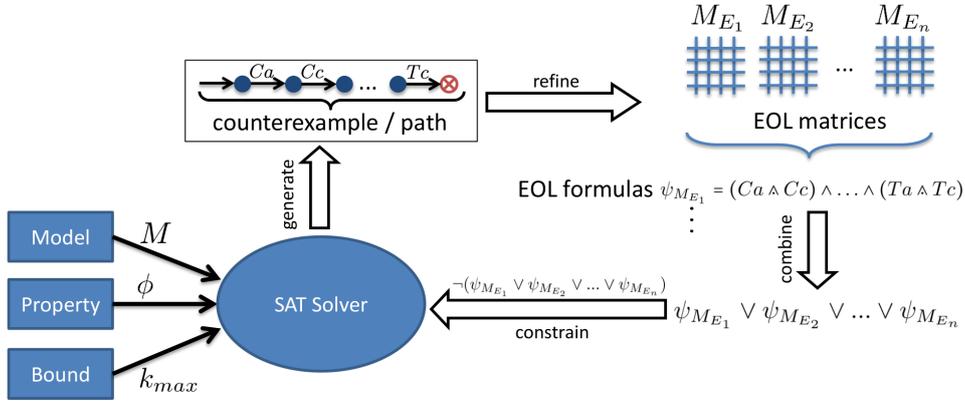


Figure 1: The iteration schema of the symbolic causality checking algorithm

3. The CX is transformed into a EOL formula in EONF and saved in a EOL Matrix.
4. The new EOL matrix is used to refine a matching, already found EOL matrix (see Definition 6) or to set up a new class of causes [LFL13c].
5. In the next iteration the event orderings in the matrices are translated into propositional logic formulas and inserted into the SAT solver in order to strengthen the constraints and, thus, find possible new orderings or new event combinations.
6. The disjunction over all EOL matrices represents the set of all computed causes of errors.

3.4 Soundness and Completeness

The following informal thoughts can be proven similar to [LFL13c]: From the definition we conclude that each CX that is found satisfies AC 1. By structural induction over the generation of the EOL matrices one can prove that new CX are always the shortest new CX that can be found and there does not exist a shorter sequence of events that lead into the property violation under the given constraints. Therefore, the minimality constraint AC 3 is fulfilled by the EOL matrices. AC 2.1 is fulfilled by each CX, since, if the last event on the CX is removed there exists a path containing a sub set of the events which does not end in a hazard state. The only problem left to solve is the AC 2.2 condition.

Event Non-Occurrence Detection. According to the AC 2.2 test the occurrence of events that are not considered as causal must not prevent the effect from happening. In other words, the non-occurrence of an event can be causal for a property violation. Therefore, we have to search such events and include their non-occurrence in the EOL formulas. In Figure 2 an example is presented which explains this procedure for an EOL formula $\psi = Ca \wedge Cc \wedge Ta \wedge Gc \wedge Tc$. Trace 1 is the minimal trace ending in a property violation. Trace 2 is non-minimal and also ends in a property violation with the events Ca, Cc, Ta, Gc, Gf, Tc . In trace 3 a new event Cl appears between Cc and Ta and no property violation is detected. This means that the appearance of the event has prevented the property violation. In order to transform this appearance into a cause for the hazard, the occurrence is negated and introduced into the EOL formula $\psi = \dots Cc \wedge_{<} \neg Cl \wedge_{>} Ta \dots$. The new clause

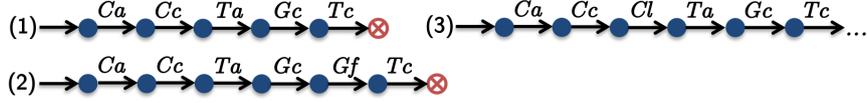


Figure 2: Three example traces for the EOL-formula $\psi = Ca \wedge Cc \wedge Ta \wedge Gc \wedge Tc$. Trace 1 is the minimal trace. While trace 2 (non-minimal) ends in a property violation, trace 3 does not.

states that “if between ‘the car is on the crossing’ and ‘the train is approaching the crossing’, ‘the car does NOT leave the crossing’, the hazard does happen”. In other words: The non-occurrence of Cl is causal for the property violation.

For every level k a second pass of the algorithm needs to be done in order to find the non-occurrences. The input parameters are altered compared to the first pass. Now the algorithm searches for paths that fulfill the property ϕ and the constraints from the EOL matrices. With this inputs the algorithm finds traces that fulfill the EOL formula and the property which must be due to an event which prevents the property violation from happening.

4 Evaluation

		Run time (sec.)			Memory (MB)		
		MC	CC 1	CC2	MC	CC1	CC 2
Railroad	explicit	0.01	0.12	0.13	16.24	16.70	17.45
	symbolic	0.01	0.16	0.51	12.36	17.61	24.86
Airbag	explicit	0.96	148.52	195.05	25.74	1,597.54	3,523.04
	symbolic	0.02	4.81	8.74	12.10	43.31	90.96

Table 1: Experimental results comparing the explicit state approach in the best case according to [LFL13a] to the symbolic approach for the railway crossing and airbag case studies.

In order to evaluate the proposed approach, we have implemented the symbolic causality checking algorithm within the symbolic model checker NuSMV2 [CCG⁺02]. Our CauSeMV extension of NuSMV2 computes the causality relationships for a given NuSMV model and an LTL property. The NuSMV models used in the experiments were generated manually. In practical usage scenarios the NuSMV models may be automatically derived from higher-level design models, as for example with the QuantUM tool [LFL11b].

As first case study we consider the railroad crossing example from Section 2.1. The second case study is the model of an industrial Airbag Control Unit taken from [AFG⁺09]. All experiments were performed on a PC with an Intel Xeon Processor (3.60 Ghz) and 144GBs of RAM. We compare our results with the results for the explicit state causality checking approach presented in [LFL13a], which were performed on the same computer.

Table 1 presents a comparison of the computational resources needed to perform the explicit and the symbolic causality checking approaches. Run. MC and Mem. MC show the runtime and memory consumption for model checking only. Run. CC1 and Mem. CC1 show the runtime and memory needed to perform causality checking without the AC2(2) condition and Run. CC2 and Mem. CC2 with the AC2(2) test enabled.

The results illustrate that for the comparatively small railroad crossing example the explicit state causality checking finishes faster and uses less memory than in the symbolic approach. For the larger airbag model the symbolic approach outperforms the explicit

approach both in terms of time and memory.

5 Related Work

In [BBDC⁺09, GMR10, GCKS06] a notion of causality was used to explain the violations of properties in different scenarios. While [BBDC⁺09, GCKS06] use symbolic techniques for the counterexample computation, they focus on explaining the causal relationships for a single counterexample and thus only give partial information on the causes for a property violation. All of the aforementioned techniques rely on the generation of the counterexamples prior to the causality analysis while our approach computes the necessary counterexamples on-the-fly. In [BV03] and [BCT07], a symbolic approach to generate Fault Trees [VGRH02] is presented. In this approach all single component failures have to be known in advance while in our approach these failures are computed as a result of the algorithm. The ordering and the non-occurrence of events can not be detected in this approach as being causal for a property violation.

6 Conclusion and Future Work

We have discussed how causal relationships in a system can be established using symbolic system and cause representations together with bounded model checking. The symbolic causality checking approach was evaluated on two case studies and compared to the explicit state causality checking approach. The symbolic causality checking can be used in an integrated tool chain, called QuantUM, in order to fully automatize the verification of systems modeled in UML / SysML and further automatically generate Fault Trees containing the causes for a system failure.

References

- [AFG⁺09] Husain Aljazzar, Manuel Fischer, Lars Grunske, Matthias Kuntz, Florian Leitner-Fischer, and Stefan Leue. Safety Analysis of an Airbag System Using Probabilistic FMEA and Probabilistic Counterexamples. In *Proc. of QEST 2009, Sixth International Conference on the Quantitative Evaluation of Systems*. IEEE Computer Society, 2009.
- [BBDC⁺09] Ilan Beer, Shoham Ben-David, Hana Chockler, Avigail Orni, and Richard Treffer. Explaining Counterexamples Using Causality. In *Proceedings of CAV 2009, LNCS*. Springer, 2009.
- [BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. *Symbolic model checking without BDDs*. Springer, 1999.
- [BCT07] M. Bozzano, A. Cimatti, and F. Tapparo. Symbolic Fault Tree Analysis for Reactive Systems. In *Proc. of ATVA 2007*, volume 4762 of *LNCS*. Springer, 2007.
- [BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.
- [BLFL14] A. Beer, F. Leitner-Fischer, and S. Leue. On the Relationship of Event Order Logic and Linear Temporal Logic. Technical Report soft-14-01, Univ. of Konstanz, Germany, January 2014. Available from: <http://www.inf.uni-konstanz.de/soft/research/publications/pdf/soft-14-01.pdf>.

- [BV03] M. Bozzano and A. Villaforita. Improving System Reliability via Model Checking: The FSAP/NuSMV-SA Safety Analysis Platform. In *Proc. of SAFECOMP 2003*, volume 2788 of *LNCS*, pages 49–62. Springer, 2003.
- [CCG⁺02] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Computer Aided Verification, 14th International Conference, CAV 2002*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002.
- [GCKS06] A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(3), 2006.
- [GMR10] Gregor Gössler, Daniel Le Métayer, and Jean-Baptiste Raclet. Causality Analysis in Contract Violation. In *Runtime Verification*, volume 6418 of *LNCS*, pages 270–284. Springer Verlag, 2010.
- [Hol03] Gerhard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison–Wesley, 2003.
- [HP05] J.Y. Halpern and J. Pearl. Causes and explanations: A structural-model approach. Part I: Causes. *The British Journal for the Philosophy of Science*, 2005.
- [Int91] International Electrotechnical Commission. Analysis Techniques for System Reliability - Procedure for Failure Mode and Effects analysis (FMEA), IEC 60812, 1991.
- [KLFL11] Matthias Kuntz, Florian Leitner-Fischer, and Stefan Leue. From Probabilistic Counterexamples via Causality to Fault Trees. In *Proceedings of the Computer Safety, Reliability, and Security - 30th International Conference, SAFECOMP 2011*, LNCS. Springer, 2011.
- [LFL11a] F. Leitner-Fischer and S. Leue. Quantitative Analysis of UML Models. In *Proceedings of Modellbasierte Entwicklung eingebetteter Systeme (MBEES 2011)*. Dagstuhl, Germany., 2011.
- [LFL11b] Florian Leitner-Fischer and Stefan Leue. QuantUM: Quantitative Safety Analysis of UML Models. In *Proceedings Ninth Workshop on Quantitative Aspects of Programming Languages (QAPL 2011)*, volume 57 of *EPTCS*, pages 16–30, 2011.
- [LFL13a] Florian Leitner-Fischer and Stefan Leue. Causality Checking for Complex System Models. In *Proc. 14th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI2013)*, LNCS. Springer, 2013.
- [LFL13b] Florian Leitner-Fischer and Stefan Leue. On the Synergy of Probabilistic Causality Computation and Causality Checking. In *Proc. of the Twentieth SPIN Symposium on Model Checking Software, SPIN 2013*, LNCS. Springer Verlag, 2013. To appear.
- [LFL13c] Florian Leitner-Fischer and Stefan Leue. Probabilistic Fault Tree Synthesis using Causality Computation. *International Journal of Critical Computer-Based Systems*, 2013. Accepted for publication.
- [Pnu77] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.
- [Sys10] Systems Modelling Language, Specification 1.2, Jun. 2010.
- [uml10] Unified Modelling Language, Specification 2.4.1, 2010.
- [VGRH02] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. *Fault Tree Handbook*, 2002.

Die Ergänzung des AUTOSAR Standards für eine durchgängige modellbasierte automobiler Steuergeräteentwicklung

Jan Meyer

Process, Methods and Tools (PMT)
HELLA KGaA Hueck & Co.
Beckumerstr. 130
59552 Lippstadt
jan.meyer@hella.com

Abstract: Bei der automobilen Steuergeräteentwicklung ist seit den letzten Jahren ein größerer Umbruch zu erkennen. Der Umbruch ist zum einen durch die steigende Komplexität der zu realisierenden Funktionen und zum anderen durch den Einsatz des domänenspezifischen AUTOSAR Standards bedingt. Dieser bringt eine Anpassung der bisherigen Entwicklung mit sich, da er bereits wesentliche Schnittstellen, Schichten, Komponenten als Vorgabe für die Softwarearchitektur definiert. Jedoch ist die Anwendung des Standards nicht ausreichend, um eine vollständige Entwicklung eines automobilen Steuergerätes vorzunehmen. Aus diesem Grund wird in diesem Papier ein Ansatz vorgestellt, wie durch den zusätzlichen Gebrauch von Modellen eine vollständige modellbasierte Entwicklung unter Berücksichtigung des AUTOSAR Standards [AUT10] erfolgen kann. Ein Schwerpunkt ist dabei die Konsistenzhaltung der unterschiedlichen Modelle, aber es wird ebenso ein Ansatz für einen (teil-) automatisierten Übergang zwischen den Modellen aufgezeigt.

1 Einleitung

Die Automobilindustrie ist derzeit in einem Wandel, der die Zukunft dieser Branche maßgeblich prägen wird. Zum Teil ist dieser Wandel auf Forderungen aus der Politik, wie eine bessere Umweltverträglichkeit (Verringerung des CO₂ Ausstoßes) oder ein besserer Verkehrsschutz (Vermeidung von Unfällen und Verminderung von Verletzungsrisiken) zurückzuführen. Um diese Anforderungen in zukünftigen Automobilen zu erfüllen, werden derzeit innovative Antriebstechniken (Elektro-, Hybrid- oder Wasserstoffantriebe) sowie Fahrerassistenzsysteme (basierend auf Radar, Ultraschall, Laser) entwickelt. Neben den politischen Bedingungen sorgt auch der Konkurrenzdruck zwischen den Zulieferern für die Entwicklung neuartiger Systeme, da die Automobilhersteller die freie Wahl zwischen den Zulieferern haben und daher immer neue Ideen gefragt sind, um auf dem hart umkämpften Markt bestehen

zu können [Bec06]. So werden beispielsweise die Komfortfunktionen, wie das Innenlicht, durch immer mehr Funktionen attraktiver gemacht. Hierdurch erhält der Kunde mehr Komfort und die Automobilhersteller ein besseres Verkaufsargument. In diesem Papier dient die Komfortlichtfunktion bei der Türsteuerung des diesbezüglichen Steuergerätes als durchgängiges Beispiel.

Bei diesen neuartigen Systemen ist eine Abkehr von der bisherigen Vorgehensweise zu sehen. Bisher war es der Fall, dass für jede neue Funktionalität ein neues Steuergerät dem Fahrzeug hinzugefügt wurde. Daher ist es auch nicht verwunderlich, dass die Anzahl der Steuergeräte auf mehr als 70 in einem Oberklassewagen gestiegen ist [NSL08]. Da eine Zunahme aus Kosten- und Platzgründen nicht mehr möglich ist, wird in dem neuartigen Systemen eine Verteilung der Funktionalität auf mehrere bereits existierende Steuergeräte angestrebt. Diese Möglichkeit war ein Grund für die Entwicklung des AUTOSAR Standards [KF09]. Förderlich ist hierbei die Weiterentwicklung der Hardware insbesondere der Rechenkerne, denn es steht für den gleichen Preis immer leistungsfähigere und oftmals mehrere Rechenkerne zur Verfügung. Dies erlaubt die Nutzung von mehr Software, bedingt aber ebenso die frühzeitige Berücksichtigung der Verteilung auf die Rechenkerne innerhalb der Entwicklung. Von daher spielt die Spezifikation des Betriebssystems eine immer größere Rolle, da die Echtzeiteigenschaften des Systems und damit die Erfüllung der zeitlichen Anforderungen hiervon abhängen.

Für die Funktionsverteilung und die damit einhergehende Spezifikation des Betriebssystems ist eine genaue Analyse der Anforderungen notwendig, um die geforderte Funktionalität sicherzustellen. Dies ist deswegen von Bedeutung, weil die automobilen Steuergeräteentwicklung geprägt ist von der Zusammenarbeit zwischen dem Automobilhersteller (OEMs) und den Zulieferern, die ihr Know-How und Wissen im Elektronikbereich in Form der Steuergeräte bereitstellen [ST12]. Damit die Integration in das Fahrzeug ohne Probleme erfolgen kann, ist eine gemeinsame Abstimmung über das zu realisierende System (Steuergerät) von Beginn der Entwicklung an notwendig. Die Abstimmung erfolgt mittels textueller Anforderungen, da diese sowohl vertragliche Grundlage sind, als auch von allen Beteiligten (Stakeholdern) gleichermaßen verstanden werden [MH13]. Da die textuellen Anforderungen auf Seiten des Automobilherstellers von mehreren unterschiedlichen Abteilungen und Personen geschrieben werden, kann es zu Inkonsistenzen und Lücken kommen. Für eine hoch qualitative Entwicklung müssen diese aber frühzeitig erkannt und behoben werden. Daher ist die Erstellung einer funktionalen Architektur ein erster Schritt auf Seiten des automobilen Zulieferers (vgl. Abbildung 1).

2 Die verschiedenen Architekturen innerhalb der Entwicklung

Bedingt durch den Wandel zu mehr komplexen Systemen sind folglich Änderungen und Anpassungen innerhalb der automobilen Steuergeräteentwicklung unausweichlich. Für die heutige automobilen Steuergeräteentwicklung spielt der AUTOSAR Standard eine zentrale Rolle, da die OEMs auf nach AUTOSAR entwickelte Steuergerä-

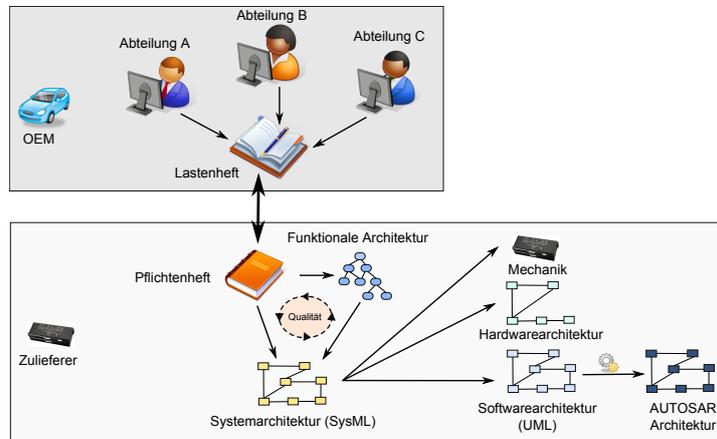


Abbildung 1: Die Artefakte bei der automobilen Steuergeräteentwicklung [MH13]

eräte bestehen. Jedoch ist die Gesamtkomplexität bei den heutigen Steuergeräten so komplex, dass sie in kleinere weniger komplexe Teilfunktionen zerteilt werden muss. Für die Modellierung einer funktionalen Architektur bietet der AUTOSAR Standard jedoch keine Möglichkeiten an, da er erst bei der Softwarearchitektur verwendet werden kann und Designentscheidungen wie Datentypen, Schnittstellen etc. voraussetzt. Von daher wird eine zusätzliche modellbasierte Modellierung benötigt, um die Dekomposition der Gesamtfunktionalität auf einzelne kleinere Unterfunktionen durchzuführen (vgl. [HHP03]). Bei der Dekomposition spielt nicht nur die Funktionshierarchie an sich eine Rolle, sondern ebenfalls die Daten- und Kontrollflüsse zwischen den (Unter-)Funktionen. Erst durch sie ist ersichtlich, welche Funktionen aus logischen und performance Gründen in einer Applikationskomponente bzw. in einer Task des Betriebssystems zusammengefasst werden. Bei dem hier vorgestellten Ansatz erfolgt die Dekomposition der Funktionen mittels der Systems Modeling Language (SysML), da mittels dieser Sprache sowohl Kontroll- als auch Datenflüsse modellierbar sind. Für die Übergänge zwischen den einzelnen Modellen (SysML, UML und AUTOSAR) (siehe Abbildung 1) wird soweit es geht auf (teil-) automatisierte Modelltransformationen zurückgegriffen. Hierdurch ist ein beispielsweise ein Übergang von der UML nach AUTOSAR unter Einbeziehung der Betriebssystemspezifikation definiert (siehe auch [HMM11] [MH11]). Bei Modellübergängen die noch Entscheidungen des Entwicklers benötigen, wie beispielsweise der Übergang von der funktionalen zur technischen Architektur innerhalb der SysML, werden Konsistenzsicherungen eingesetzt, die auf der manuell erstellen Nachverfolgbarkeit (Traceability) aufbauen und automatisch die Konsistenz zwischen den Modellen sicherstellen. Hierdurch entsteht am Ende ein AUTOSAR konformes Modell, aber es wird zusätzlich auf eine modellbasierte funktionale Architektur mit ihren Vorteilen für die Entwicklung zurückgegriffen.

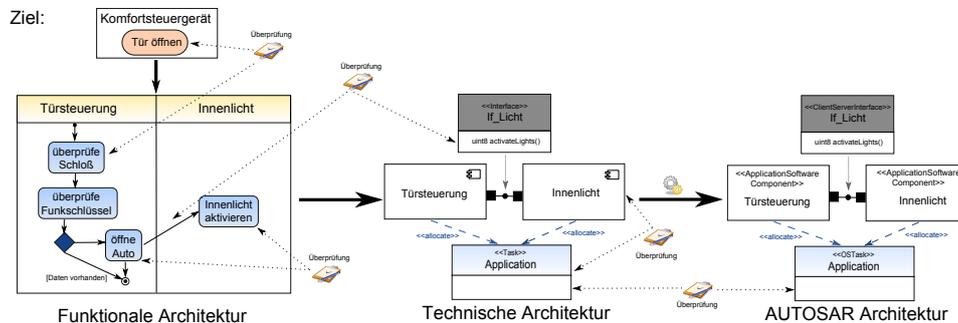


Abbildung 2: Beispiel der Konsistenzsicherung zwischen funktionaler und technischer Architektur

2.1 Die funktionale Architektur

In der funktionalen Architektur wird beschrieben, was das System machen wird, aber nicht, wie es die Funktionen umsetzt. Dies erfolgt in der technischen Architektur [KSW13]. Für die funktionale Architektur werden die Ergebnisse der Analyse der textuellen Anforderungen verwendet. Zunächst werden die Ziele des Systems festgelegt (siehe Abbildung 2). Anschließend werden die Ziele in Funktionen und Unterfunktionen aufgeteilt. Dabei werden die einzelnen Funktionen des Systems und ihre Kommunikation untereinander spezifiziert. Hieraus ist ersichtlich, welche Funktionen zueinander in Verbindung stehen und daher Kommunikationskanäle innerhalb der technischen Architektur benötigen. Die funktionale Architektur ist dabei eine sehr stabile Grundlage, da an der Funktionalität nur wenige Änderungen vorgenommen werden (siehe [LW10]). Sie ist außerdem Basis für die Erstellung des funktionalen Sicherheitskonzeptes für sicherheitskritische Systeme nach ISO 26262 [ISO11] [LPP10]. Den einzelnen Funktionen und ihren Verbindungen lassen sich Sicherheitsziele und dementsprechend Sicherheitseinstufungen (ASIL-Level) zuordnen. Die Sicherheitseinstufung wird ebenso in der technischen Architektur benötigt, um die konkreten Maßnahmen zur Absicherung von Daten und Algorithmen vorzunehmen. Sie werden mittels einer Erweiterung der SysML durch Eigenschaftswerte (Tagged Values) spezifiziert. Wie hierbei bereits zu erkennen ist, spielt die Konsistenzsicherung zwischen den beiden Architekturen eine besondere Rolle (vgl. Kapitel 2.3).

Zum besseren Verständnis wird anhand eines Beispiels, in diesem Fall eines Komfortsteuergerätes, welches neben der Türsteuerung ebenso für die Lichtsteuerung zuständig ist, die funktionale Architektur beschrieben. Zunächst gilt es die Ziele des Steuergerätes zu erfassen und zu modellieren. In diesem Beispiel geht es um die Steuerung der Lichtfunktion beim Öffnen der Tür. Dies Ziel wird mittels eines Anwendungsfalls dargestellt (siehe Abbildung 2). Es wird anschließend in einzelne Funktionen dekomponiert, um die notwendigen Funktionen zu erkennen, die für die Realisierung des Ziels notwendig sind. Dies erfolgt mit Hilfe von Aktivitätsdiagramm-

men (vgl. Abbildung 2). Die Kommunikation und die Abhängigkeiten untereinander werden in Form von Kontroll- und Datenflüssen beschrieben. Zur Erstellung der funktionalen Architektur werden die Funktionen funktionalen Blöcken, den sogenannten «FunctionalBlock» zugeordnet. Diese werden später den Teilsystemen der technischen Architektur zugewiesen.

2.2 Die technische Architektur

In der technischen Architektur werden die logischen Teilsysteme und die physikalischen Elemente und ihr Zusammenspiel spezifiziert. Hierbei wird zunächst mit der Systemarchitektur begonnen. In dieser Architektur werden die verschiedenen Disziplinen gleichberechtigt dargestellt, d. h. Elektronik, Informatik und Mechanik werden innerhalb der Systemarchitektur beschrieben. Dabei wird die Systemarchitektur soweit verfeinert, bis das Teilsystem einer Disziplin zugeordnet werden kann. In diesem Fall durch einen Stereotype für die Blöcke der SysML («Software», «Hardware», «Mechanic»). Anschließend wird in die disziplinspezifische Entwicklung übergegangen. Für die Software ist dies die Softwarearchitektur. In dieser wird die Struktur der Software, die Kommunikation und das Verhalten festgelegt. Da insbesondere für das Verhalten der AUTOSAR Standard keine Modellierungsmöglichkeit aufweist, wird zunächst die Softwarearchitektur in UML erstellt. Hierbei werden die für den AUTOSAR notwendigen Informationen bereits berücksichtigt. So wird eine Komponente mittels neu definierter Stereotypen bereits als Applikations-, Service- oder Basiskomponente bzw. als ComplexDeviceDriver gekennzeichnet.

Für den Übergang von der UML nach AUTOSAR werden vom Entwickler keine zusätzlichen Entscheidungen benötigt, da alle notwendigen Informationen bereits in der UML Architektur vorliegen. Von daher bietet sich bei diesem Übergang die Verwendung einer Modelltransformation an [MH11]. Die Transformation stellt nicht nur sicher, dass die Daten wieder verwendet werden können und konsistent sind, sondern sie stellt ebenso eine notwendige Nachverfolgbarkeit (Traceability) dar. Bei der hier verwendeten Transformation werden jedoch nicht nur die Komponenten berücksichtigt, sondern es wird ebenso die Spezifikation des Betriebssystems berücksichtigt, da es große Auswirkungen auf die Echtzeitfähigkeit aufweist. Infolge der Modelltransformation verbindet dieser Ansatz die zusätzlichen Modellierungsmöglichkeiten der UML mit der notwendigen Verwendung des AUTOSAR Standards.

2.3 Die Transformation bzw. Konsistenzhaltung der Architekturen

Für die automobilen Steuergeräteentwicklung ist es wichtig, dass die einzelnen Modelle konsistent zueinander sind, besonders da häufig durch Änderungen vom OEM die Architektur angepasst werden muss. Dabei ist es notwendig nicht nur die

technische, sondern auch die funktionale Architektur anzupassen, denn von der funktionalen Architektur ist beispielsweise das funktionale Sicherheitskonzept und damit auch die späteren Absicherungen für sicherheitskritische Systeme abhängig. Von daher wäre es fatal, wenn die funktionale und die technische Architektur inkonsistent zueinander sind. Genauso katastrophal wäre es, wenn die technische Architektur in sich nicht stimmig ist und daher die Transformation abbrechen würde. Dann könnte kein AUTOSAR konformes System erstellt werden. Damit die Modelltransformation oder die Konsistenzsicherung erfolgen kann, ist die Vollständigkeit und Korrektheit der Modelle Grundvoraussetzung.

Bei dem Übergang von der funktionalen zur technischen Architektur kann nicht auf Modelltransformationen zurückgegriffen werden, da hier der Architekt noch Entscheidungen treffen muss und daher keine automatisierte Transformation möglich ist. Von daher muss die Konsistenz auf einem anderen Wege sichergestellt werden.

Um Korrektheit und die Vollständigkeit der Modelle sicherzustellen, ist zunächst zu spezifizieren, welche Elemente und in welcher Anzahl in den Modellen vorkommen. Zu diesem Zweck ist für jede Architektur eine (semi-) formale Definition der Methodik mittels der UML erfolgt. Hierdurch ist eindeutig beschrieben, welche Elemente und wie häufig in dem Modell vorkommen dürfen bzw. müssen. Anhand des Beispiels (siehe Abbildung 2) des Komfortsteuergerätes ist das Ergebnis der Methodik bezüglich der Modellierung der funktionalen Architektur dargestellt. Das Beispiel zeigt aber noch nicht, wie die einzelnen Bestandteile der funktionalen Architektur zusammenwirken bzw. wie sie bei einer anderen funktionalen Architekturen aussehen. Von daher ist eine (semi-) formale Definition der Methodik notwendig (vgl. Abbildung 3). In Auszug der Definition der funktionalen Architektur ist zu erkennen, dass sie aus Zielen, Funktionen und Funktionsblöcken besteht und von welchen Meta-Elementen aus der SysML diese abgeleitet sind. Ebenso ist ein Auszug aus den Elementen der technischen Architektur zu sehen.

Die formale Definition der Methodik hilft zum einen bei der Aus- und Weiterbildung der Entwickler, da sie die Zusammenhänge zwischen den einzelnen Elementen aufzeigt. Zum anderen kann sie herangezogen werden, um Überprüfungen davon abzuleiten. Mittels der Überprüfungen wird in einem Review sichergestellt, dass das entwickelte Modell der Entwicklungsmethodik entspricht und das somit die Modelle korrekt und vollständig sind. Da manuelle Reviews einen großen Aufwand mit sich bringen, ist es sinnvoll möglichst viele Fragestellungen durch automatisierte Überprüfungen (Checks) zu kontrollieren.

Aus diesem Grund wurde parallel zur Spezifikation der Methodik beispielsweise für die funktionale Architektur automatisierte Überprüfungen erarbeitet, die sicherstellen, dass sowohl die Modelle korrekt und vollständig sind, aber auch die Konsistenz zur technischen Architektur korrekt ist. Denn zwischen beiden muss die Konsistenz überprüft werden, da ein manueller Übergang genutzt wird. Dabei werden Regeln für die Einhaltung der Methodik als auch zur Sicherstellung der Konsistenz abgefragt. Exemplarische Regeln sind in der folgenden Aufzählung aufgeführt.

- Eine Abhängigkeit in der funktionalen Architektur in verschiedene Aktivitäts-

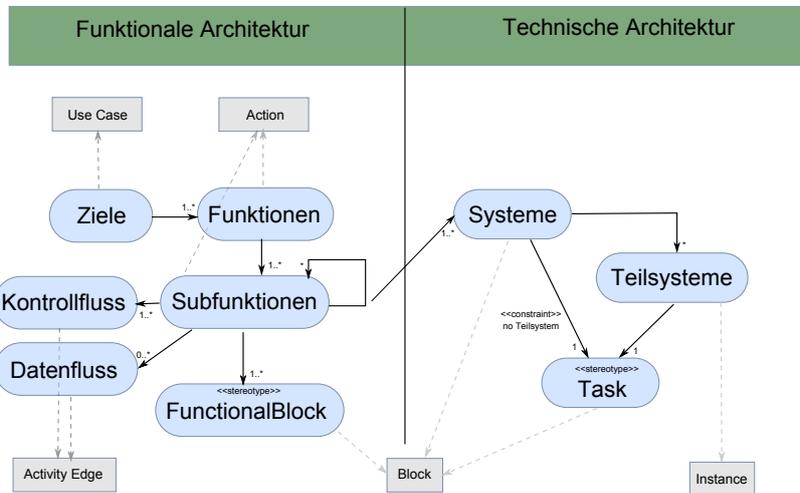


Abbildung 3: Auszug aus der Spezifikation der funktionalen Architektur

bereiche bedingt eine Schnittstelle zwischen den beiden Teilsystemen (*Übergang funktionale zur technischen Architektur*)

- Ein funktionaler Block ist mindestens einem System zugeordnet (*Übergang funktionale zur technischen Architektur*)
- Ein System oder Teilsystem muss immer einer Task zugeordnet sein (*technische Architektur*)
- Eine Task muss immer eine Priorität besitzen (*technische Architektur*)
- Ein Ziel muss mindestens einer Funktion zugeordnet sein (*Übergang Ziel funktionale Architektur*)
- Eine Funktion ist mindestens einem funktionalen Block zugeordnet (*funktionale Architektur*)
- Eine Schnittstelle darf immer nur Attribute oder Operationen beinhalten (*Übergang funktionale Architektur zu AUTOSAR*)

Das automatisierte Vorgehen ist nur deshalb möglich, weil die Informationen in (semi-) formaler Form vorliegen. Dies ist ein großer Vorteil der modellbasierten Entwicklung. Die Regeln wurden dabei mittels der Programmiersprache Java realisiert, da diese von dem eingesetzten Werkzeug¹ für die UML und SysML Erstellung unterstützt wird. Die Regeln sind dabei im Tool an sich integriert und werden zu den Standard Überprüfungen des Tools ausgeführt (vgl. Abbildung 4). Der Benutzer muss somit nicht an unterschiedlichen Stellen nach Ergebnissen schauen

¹IBM Rational Rhapsody - <http://www-03.ibm.com/software/products/en/ratirhapfami>

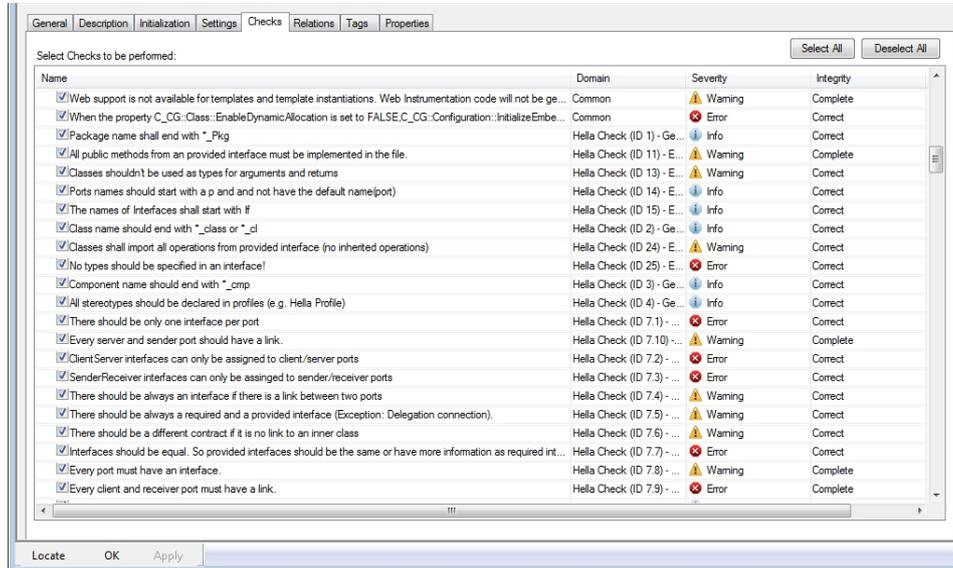


Abbildung 4: Abbild der implementierten Überprüfungen

und wird auch bei Fehlern direkt zu dem jeweiligen Modellelement navigiert, so dass er den Fehler schnellstmöglich beheben kann.

3 Verwandte Arbeiten

Die Wahl zur Modellierung der funktionalen Architektur ist bei dem hier vorgestellten Ansatz auf die Systems Modeling Language (SysML) bzw. die Unified Modeling Language (UML) gefallen. Hauptgründe waren die weltweite Bekanntheit (internationaler Standard), die Modellierung und Dokumentation einer vollständigen Systemarchitektur, die Verfügbarkeit von Werkzeugen und die Erweiterungsmöglichkeit, so dass die Modellierung dem Entwicklungsprozess und vor allem der funktionalen Architektur angepasst werden kann. Aus den eben genannten Gründen sind andere modellbasierte Ansätze wie EAST-ADL [Ead08] (keine Werkzeugunterstützung), Matlab/Simulink² (keine Architekturmodellierung) oder aber textuelle Satzmuster [HMvD11] (keine graphische Darstellung und somit eine fehlende Abstraktion und Übersicht) nicht zum Einsatz gekommen.

Für die funktionale Modellierung existieren bereits einige Arbeiten, die ebenfalls die SysML bzw. UML verwendet haben (siehe beispielsweise [RBvdBS02] [Mut05]). In diesen Arbeiten wird zwar beschrieben, wie eine funktionale Architektur mit UML aussehen kann, jedoch wird entweder keine Anpassung an die Automobilin-

²siehe <http://www.mathworks.de/products/simulink/>

dustrie vorgenommen [Mut05] oder aber es werden keine automatisierten Überprüfungen eingesetzt [RBvdBS02], um eine konsistente und vollständige Architektur für eine Transformation zu haben

Bezüglich der Verbindung von AUTOSAR mit weiteren modellbasierten Ansätzen existieren ebenso verwandte Arbeiten. Exemplarisch kann auf [GHN09] und darauf aufbauende Arbeiten verwiesen werden. In dieser Arbeit wird mittels einer Modelltransformation die Daten aus einem SysML Modell nach AUTOSAR und zurück gebracht. Der Übergang von SysML nach AUTOSAR ist unserer Meinung nach jedoch noch zu früh, da in der Systemarchitektur mit SysML oftmals noch keine Datentypen und Schnittstellen spezifiziert sind. Diese Informationen sind für den AUTOSAR Standard zwingend erforderlich. Ebenso betrachtet der Ansatz nur einen Modellübergang für die Applikationsebene. Die ebenso für ein automobiles Steuergerät wichtige Spezifikation des Betriebssystems wurde jedoch außer Acht gelassen. Von daher wurde für diese Arbeit der Ansatz aus [GHN09] als Grundlage genommen und für den Einsatz in automobilen Serienprojekten angepasst.

4 Zusammenfassung und Ausblick

In diesem Beitrag wurden die Vorteile der modellbasierten Entwicklung eines automobilen Steuergerätes beschrieben. Schwerpunkt war die Modellierung der funktionalen Architektur, die Konsistenzsicherung zur technischen Architektur und die Integration des AUTOSAR Standards. Hiermit ist ein erster Schritt in Richtung zur vollständigen modellbasierten Entwicklung im Automobilbereich gegeben. Durch die neuartigen Systeme werden vermehrt sicherheits- und zeitkritische Funktionen realisiert. Diese benötigen eine besondere Analyse und Umsetzung gemäß dem neuen Standard ISO 26262. Die notwendigen Daten sind bereits in der Architektur vorhanden und müssen in die entsprechenden Werkzeuge übertragen werden. Hierzu kann eine ähnliche Modelltransformation wie beim Übergang nach AUTOSAR verwendet werden. Als Ergebnis entsteht ein modellbasierter Ansatz aus dem sich die verschiedensten Werkzeuge bedienen und eine Nachvollziehbarkeit (Traceability) herstellen.

Literatur

- [AUT10] AUTOSAR Gbr. Version 4.0. AUTOSAR Standard, 2010.
- [Bec06] Helmut Becker. *High Noon in the Automotive Industry*. Springer Verlag, 2006.
- [Ead08] EAST-ADL2 Specification, 2008.
- [GHN09] Holger Giese, Stephan Hildebrandt, and Stefan Neumann. Towards Integrating SysML and AUTOSAR Modeling via Bidirectional Model Synchronization. In Holger Giese, Michaela Huhn, Ulrich Nickel, and Bernhard Schätz,

- editors, *Tagungsband Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme*, pages 155 – 164. Informatik-Bericht 2009-01 TU Braunschweig, Institut für Software Systems Engineering Technische Universität Braunschweig, 2009.
- [HHP03] Derek Hatley, Peter Hruschka, and Imtiaz Pirbhai. *Komplexe Software-Systeme beherrschen*. mitp-Verlag, 2003.
- [HMM11] Jörg Holtmann, Jan Meyer, and Matthias Meyer. A Seamless Model-Based Development Process for Automotive Systems. In *Software Engineering 2011 Ü Workshopband (inkl. Doktorandensymposium)*, volume P-184. Bonner Köllen Verlag, GI-Edition Lecture Notes in Informatics (LNI), 2011.
- [HMvD11] Jörg Holtmann, Jan Meyer, and Markus von Detten. Automatic Validation and Correction of Formalized, Textual Requirements. In *Proceedings of the IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW) 2011*, 2011.
- [ISO11] ISO 26262 Road vehicles - Functional safety, 2011.
- [KF09] Olaf Kindel and Mario Friedrich. *Softwareentwicklung mit AUTOSAR*. dpunkt-Verlag, 2009.
- [KSW13] Rüdiger Kaffenberger, Sven-Olaf Schulze, and Hanno Weber. *INCOSE Systems Engineering Handbuch*. Hanser Verlag, 2013.
- [LPP10] Peter Löw, Roland Pabst, and Erwin Petry. *Funktionale Sicherheit in der Praxis*. dpunkt.verlag, 2010.
- [LW10] Jesko G. Lamm and Tim Weilkens. Funktionale Architekturen in SysML. In Maik Maurer and Sven-Olaf Schulze, editors, *Tag des Systems Engineering (TdSE 2010)*, pages 109 –118. GfSE, Hanser Verlag, 2010.
- [MH11] Jan Meyer and Jörg Holtmann. Eine durchgängige Entwicklungsmethode von der Systemarchitektur bis zur Softwarearchitektur mit AUTOSAR. In Holger Giese; Michaela Huhn; Jan Philipps; Bernhard Schätz, editor, *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VII*, pages 21–30. fortiss GmbH, 2011.
- [MH13] Jan Meyer and Wilfried Horn. Modellbasiertes Systemengineering zur Qualitätsverbesserung bei der Entwicklung eines automobilen Steuergerätes. In Maik Maurer and Sven-Olaf Schulze, editors, *Tag des Systems Engineering (TdSE 2013)*, pages 315 – 324. Gesellschaft für Systems Engineering, Carl Hanser Verlag, 2013.
- [Mut05] Martin Mutz. *Eine Durchgängige modellbasierte Entwurfsmethodik für eingebettete Systeme im Automobilbereich*. Cullivier Verlag Göttingen, 2005.
- [NSL08] Nicolas Navet and Françoise Simonot-Lion. *Automotive Embedded Systems Handbook*. CRC Press, 2008.
- [RBvdBS02] Martin Rappl, Peter Braun, Dr. Michael von der Beeck, and Dr. Christian Schröder. Automotive Software Development: A Model Based Approach. *Society of Automotive Engineers, Inc.*, 2002.
- [ST12] T. Streichert and M. Traub. *Elektrik/Elektronik-Architekturen im Kraftfahrzeug: Modellierung und Bewertung von Echtzeitsystemen*. Springer Verlag, 2012.

Eine Anleitung zur Entwicklung von Simulink-Targets für die Lehre

Frank Tränkle

Automotive Systems Engineering
Hochschule Heilbronn
Max-Planck-Str. 39
D-74081 Heilbronn
frank.traenkle@hs-heilbronn.de

Zusammenfassung: Die modellgetriebene Softwareentwicklung mit Simulink oder ASCET ist in den Studiengängen Automotive Systems Engineering, Elektrotechnik, Antriebssysteme und Mechatronik oder in vergleichbaren Studiengängen erfolgreich eingeführt. Allerdings schreitet die durchgängige Anwendung von Simulink aufgrund mangelnder Verfügbarkeit geeigneter Simulink-Targets für Laborversuche und Hochschulprojekte nur langsam voran.

Dieser Beitrag beschreibt Vorgehensweisen und Architekturen, wie Simulink-Targets für ARM-Cortex-M-basierte elektronische Steuergeräte erfolgreich entwickelt werden können, so dass sie den Anforderungen für den Einsatz in der Lehre genügen. Dabei werden verschiedene existierende Simulink-Targets mit dem neuen Simulink-Target MB-BOX32 verglichen, die sich sowohl in den Schnittstellen zwischen Basis-Software und Simulink-Modell als auch in der benötigten Mathworks-Software unterscheiden.

Keywords: Simulink-Target, Simulink Coder, Embedded Coder, modellgetriebene Softwareentwicklung, modellbasierte Softwareentwicklung, ARM-Cortex-M

1 Einleitung

Die modellgetriebene Entwicklung von Embedded Software ist von entscheidender Bedeutung für die Realisierung von Innovationen im Automobilbereich [BR11] und in anderen Branchen. Automotive Systems Engineering, Elektrotechnik, Antriebssysteme und Mechatronik oder vergleichbare Studiengänge bilden Studierende für eine spätere berufliche Tätigkeit im Bereich der modellgetriebenen Softwareentwicklung aus. Der Einsatz von Simulink oder ASCET in der Lehre wurde dabei erfolgreich etabliert.

Jedoch schreitet die durchgängige Anwendung von Simulink aufgrund mangelnder Verfügbarkeit von Simulink-Targets für Laborversuche und Hochschulprojekte langsam voran. Vielfach werden zur Steuerung und Regelung von Laborversuchen Rapid-

Prototyping-Systeme eingesetzt. Für eine praxisnahe Ausbildung sind aber seriennahe 32-bit-Steuergeräte besser geeignet.

Im Studiengang Automotive Systems Engineering an der Hochschule Heilbronn werden ARM-Cortex-M-basierte Steuergeräte für regelungstechnische Laborversuche verwendet. Dabei kommen STM32-Microcontroller von STMicroelectronics als eine einheitliche Microcontroller-Plattform zum Einsatz. Für diese Plattform wird ein neues Simulink-Target MB-BOX32 entwickelt.

Der folgende Abschnitt 2 spezifiziert zunächst die Anforderungen an ein Simulink-Target für den Einsatz von 32-bit-Microcontrollern im Labor Regelungstechnik. Abschnitt 3 gibt einen Überblick über bereits existierende Simulink-Targets für Cortex-M-Microcontroller. In Abschnitt 4 werden die Architektur und Schnittstellen der Applikations- und Basis-Software für das neue Simulink-Target MB-BOX32 vorgestellt und mit den Simulink-Targets aus Abschnitt 3 verglichen.

2 Anforderungen

Ein Simulink-Target für die Lehre muss äußerst schnell erlernbar sein, modular aufgebaut sein und aus kostengünstigen Hardware- und Softwarekomponenten bestehen. Existierende Hardware und Software muss ohne großen Aufwand an neue Laborversuche angepasst werden können. Als wesentliche Anforderung ist daher eine Verfügbarkeit aller Quellen des Simulink-Targets und der Basis-Software erforderlich.

Die Applikations-Software wird in Simulink modelliert und umfasst die Steuerungs-, Regelungs-, Signalverarbeitungs- und Überwachungsfunktionen. Für den Zugriff der Applikations-Software auf die Steuergeräte-I/O werden die Schnittstellen der Basis-Software als Ein- und Ausgangssignale in Simulink zur Verfügung gestellt. Damit können sich die Studierenden im Labor Regelungstechnik auf die Modellierung der Applikations-Software konzentrieren.

Die Basis-Software konfiguriert das Steuergerät, enthält einen Scheduler zur Ausführung der Funktionen der Applikations-Software und greift auf die I/O-Schnittstellen und Module des Microcontrollers zu. Die Basis-Software wird speziell für den jeweiligen Laborversuch angepasst. Sie kann optional in Simulink modelliert oder von Hand in C und Assembler programmiert werden.

Für die Online-Messung und -Parameterstellung muss das Steuergerät über kabelgebundene und drahtlose Applikations-Schnittstellen verfügen. Als Applikationswerkzeuge kommen CANape oder alternativ INCA zum Einsatz. Dazu muss das Simulink-Target die ASAP2-Generierung für Signale und Parameter unterstützen.

In einer ersten Stufe wird als Toolchain Simulink mit Simulink Coder und optional Embedded Coder eingesetzt. Für zukünftige Projekte soll jedoch auch ein Einsatz von

TargetLink oder ASCET möglich sein. Die Basis-Software mit ihren Schnittstellen soll ohne Modifikationen wiederverwendet werden können.

3 Stand der Technik

Für STM32-Microcontroller von STMicroelectronics steht bereits eine Reihe von Simulink-Targets zur Verfügung, die hier an dieser Stelle in einer Kurzübersicht beschrieben werden.

Waijung-Blockset

Beim Waijung-Blockset handelt sich um ein umfassendes Simulink-Embedded-Coder-Target für verschiedene Cortex-M-Varianten aus der STM32-Familie [WA14]. Dieses Target generiert sowohl den Applikations-Code als auch die Basis-Software aus Simulink.

Die Konfiguration des Microcontrollers und Basis-Software erfolgt über Simulink-Blöcke. Zur Ansteuerung der Microcontroller-Module steht ein umfassendes Simulink-Blockset zur Verfügung. Unterstützt werden Simulink-Modelle mit mehreren Zeitscheiben sowie Interrupt-Routinen. Die Zeitscheiben werden sequentiell in einer Task gerechnet (Multi-Rate Single-Task-Mode).

Als Toolchain zur Compilierung des erzeugten Codes enthält das Waijung-Blockset die GNU-ARM-Toolchain. Eine Integration des erzeugten Codes in einer Entwicklungsumgebung ist nicht direkt vorgesehen, ist aber manuell möglich.

Das Waijung-Blockset ist ausschließlich über neue Blöcke erweiterbar. Existierende Blöcke können nicht verändert werden. Die vorhandenen Simulink-Blöcke und MATLAB-Routinen stehen nur als Binär-Code zur Verfügung.

Embedded Coder Support Packages ARM Cortex-M3- und STM32F4-Discovery

The Mathworks bietet Support Packages an für Rapid-Prototyping-Anwendungen auf verschiedenen Target-Plattformen, so auch für Cortex-M3- und Cortex-M4-Microcontroller aus der STM32-Familie [MW14].

Im Gegensatz zu anderen Target Support Packages sind für Cortex-M3 und Cortex-M4 Lizenzen für Simulink und Embedded Coder erforderlich. Damit fällt ein wichtiger Vorteil anderer Target Support Packages weg, die nur eine Installation von MATLAB und Simulink voraussetzen und daher besonders interessant für Studierende sind.

Ähnlich wie beim Waijung-Blockset erfolgt sowohl die Generierung des Applikations-Codes als auch der Basis-Software vollständig aus Simulink. Im Gegensatz zum Waijung-Blockset ist der Funktionsumfang des Simulink-Blocksets zur Konfiguration des Microcontrollers und zur Ansteuerung der I/O sehr eingeschränkt.

Als Targets werden QEMU, eine Emulationsumgebung für Cortex-M3-Controller, sowie das Evaluations-Board STM32F4-Discovery unterstützt. Für STM32F4-Discovery stehen Simulink-Blöcke für ADC und GPIO zur Verfügung. Zur Compilierung wird die GNU-ARM-Toolchain verwendet. Eine Integration des Codes in eine Entwicklungsumgebung ist manuell möglich.

Eine Erweiterung des Targets für andere Steuergeräte wird dadurch erschwert, dass die Simulink-Blöcke und MATLAB-Routinen nur als Binär-Code zur Verfügung stehen. Daher kann das Blockset auch nur durch neue Blöcke erweitert werden. Existierende Blöcke können nicht verändert werden.

STM32-MAT/TARGET von STMicroelectronics

Das neueste Target in dieser Reihe wird von STMicroelectronics zur Verfügung gestellt [ST14]. Der Funktionsumfang ist sehr ähnlich zum Funktionsumfang des Support Packages von The Mathworks. Es dient ebenfalls zur Codegenerierung für das Evaluationsboard STM32F4-Discovery. Die Erweiterbarkeit ist ähnlich eingeschränkt. Allerdings werden neben ADC und GPIO Timer und UART unterstützt.

Voraussetzung für die Codegenerierung sind installierte Lizenzen des Simulink Coders und Embedded Coders. Als wesentlicher Unterschied zum Support Package von Mathworks wird der von STM32-MAT/TARGET erzeugte C-Code in die Entwicklungsumgebungen Keil μ Vision, IAR EWARM oder Atollic True-Studio integriert.

Fazit

Da die oben beschriebenen Simulink-Targets nur schwer anpassbar und erweiterbar sind, werden diese im Labor Regelungstechnik des Studiengangs Automotive Systems Engineering nicht eingesetzt. Auch die notwendige Installation des Embedded Coders widerspricht der Anforderung an eine möglichst kostengünstige Lösung.

Aus diesen Gründen wird ein neues Simulink-Target namens MB-BOX32 für den Einsatz im Labor Regelungstechnik entwickelt, dessen Funktionsweise und Architektur im folgenden Abschnitt beschrieben wird.

4 Architektur von MB-BOX32

Das neue Simulink-Target MB-BOX32 verfügt im Gegensatz zu den existierenden Simulink-Targets [MW14], [ST14] und [WA14] über keinen Simulink-Blockset für den Zugriff auf die Steuergeräte-I/O. Vielmehr erfolgt die Kommunikation zwischen Applikations-Modell und der Basis-Software über Signalschnittstellen ähnlich wie bei AUTOSAR.

Während die Simulink-Targets [MW14], [ST14] und [WA14] eine Installation des Embedded Coders zusätzlich zum MATLAB Coder und Simulink Coder voraussetzen,

genügt für MB-BOX32 eine Installation des MATLAB Coders und Simulink Coders, was die Softwarelizenzkosten reduziert.

Die folgende Beschreibung der Architektur des Applikations-Modells, des Simulink-Targets und der Basis-Software ist eine Anleitung für die Entwicklung weiterer Simulink-Targets für 32-bit-Microcontroller. Denn die Ansätze sind nicht auf Cortex-M-basierte Steuergeräte beschränkt sondern universell anwendbar.

Architektur des Applikations-Modells

Die Kommunikation zwischen Applikations-Modell und der Basis-Software erfolgt über eine Signalschnittstelle. Die Simulink-I/O-Signale werden als MATLAB-Workspace-Variablen definiert und den Signallinien im Simulink-Blockschaltbild über Properties zugeordnet. Das Simulink-Target MB-BOX32 erzeugt bei der Codegenerierung für diese Signallinien Schreibe- und Leseanweisungen auf globale Variablen. Diese globale Variablen werden wiederum von den I/O-Routinen der Basis-Software gelesen und geschrieben. Dadurch erfolgt der Signalaustausch zwischen Applikations-Software und Basis-Software.

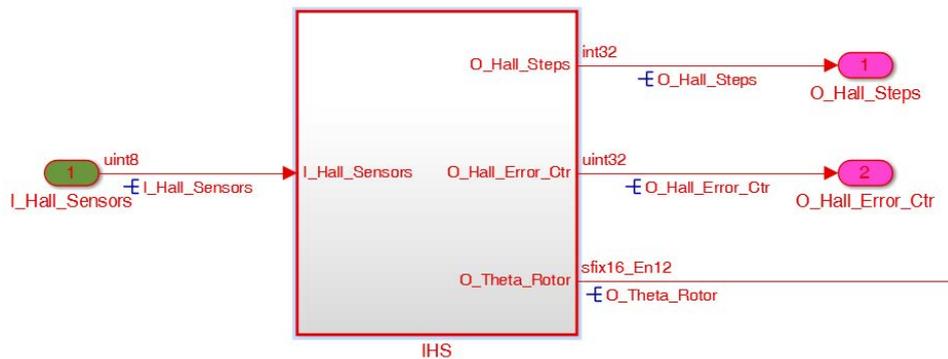


Abbildung 1: Zuordnung von im MATLAB-Workspace definierten Simulink-Signalen zu Signallinien.

Die einzelnen Signallinien werden im Falle von Eingangssignalen durch Inport-Blöcke und bei Ausgangssignalen durch Outport-Blöcke abgeschlossen. Über diese Inport- und Outport-Blöcke kann das Applikations-Modell zusammen mit Strecken-Modellen zu System-Modellen integriert werden. Ohne Modifikationen kann dann das Applikations-Modell entweder zur Codegenerierung verwendet oder in MiL- und SiL-Tests simuliert werden. Die Implementierung des Applikations-Modells kann dabei entweder als Simulink-Subsystem oder als eigenständiges Simulink-Modell erfolgen.

Abbildung 1 zeigt einen Ausschnitt eines Applikations-Modells zur feldorientierten Regelung eines bürstenlosen Gleichstrommotors in einem Laborversuch. Die Codegenerierung bildet die Signallinien I_Hall_Sensors, O_Hall_Steps, O_Hall_Error_Ctr und O_Theta_Rotor in Lese- bzw. Schreibzugriffe auf globale

Variablen ab. Über die dargestellten Inports und Outports wird das Applikations-Modell mit einem Streckenmodell für den Gleichstrommotor verschaltet.

Simulink-Target

Für jedes Laborprojekt wird ein individuelles MATLAB-M-Skript zur Verfügung gestellt, das die Simulink-I/O-Signale als Workspace-Variablen definiert. Damit wird sichergestellt, dass das Applikations-Modell nur auf tatsächlich vorhandene I/O zugreift. Weiterhin wird eine einfache Anpassbarkeit der Schnittstelle erzielt. Abbildung 2 zeigt als Beispiel die Definition des Simulink-Signals I_Hall_Sensors als MATLAB-Workspace-Variable.

```
I_Hall_Sensors = mb32.Signal;  
I_Hall_Sensors.Description = 'Rotor position information';  
I_Hall_Sensors.DataType = 'uint8';  
I_Hall_Sensors.CoderInfo.StorageClass = 'Custom';  
I_Hall_Sensors.CoderInfo.CustomStorageClass = 'mb32_IO';
```

Abbildung 2: Definition des Simulink-Signals I_Hall_Sensors als Workspace-Variable zur Verwendung im Applikations-Modell.

Das Simulink-Target MB-BOX32 unterstützt Multi-Rate-Simulink-Modelle sowohl im Single-Tasking- als auch Multi-Tasking-Mode. Für die Verarbeitung von asynchronen Interrupts wird ein Simulink-Block zur Verfügung gestellt, der den Interrupt Service Request im Applikations-Modell als asynchronen Funktionsaufruf abbildet. Interrupt-Service-Routinen können so als Function-Call-Subsysteme modelliert werden.

Der vom Simulink Coder erzeugte Code wird in der Entwicklungsumgebung Keil μ Vision mit der Basis-Software und der Bibliothek CMSIS von STMicroelectronics integriert. Dadurch wird bei Bedarf ein direktes Debugging der Applikations- und Basis-Software ermöglicht.

MB-BOX32 setzt eine Installation des Simulink Coders voraus und unterstützt optional den Embedded Coder. Ein wesentlicher Unterschied zwischen dem Basisprodukt Simulink Coder und der Erweiterung Embedded Coder besteht darin, dass der Embedded Coder Custom-Storage-Classes unterstützt. Durch die Verwendung von Custom-Storage-Classes bei der Schnittstellen-Definition wird eine prozesssichere Integration mit der Applikations-Software mit der Basis-Software erzielt.

Architektur der Basis-Software

Die Basis-Software besteht aus den folgenden Modulen: Scheduler, Hardwareabstraktionsschicht CMSIS, Initialisierungsroutine, I/O-Routinen, XCP-Server. Der Scheduler wird durch den Cortex-M-System-Timer getaktet. Im Falle des Single-Tasking-Modes wird eine vom Simulink Coder erzeugte C-Funktion mit der Basisrate des Simulink-Modells aufgerufen. Beim Multi-Tasking-Mode werden vom Scheduler Tasks erzeugt und aufgerufen, die den einzelnen Zeitscheiben des Simulink-Modells entsprechen.

Der standardmäßige Scheduler hat ein minimales Footprint und unterstützt nicht-preemptives Multi-Tasking. Durch einen optionalen Einsatz von FreeRTOS kann preemptives Multi-Tasking realisiert werden.

Über Hook-Funktionen ruft der Scheduler Routinen zum Lesen und Schreiben der I/O-Signale auf, die für den jeweiligen Laborversuch in C entwickelt werden. Auch der XCP-Server wird über eine Hook-Funktion zyklisch aufgerufen. Die Konfiguration des Steuergeräts inklusive Microcontroller erfolgt vor dem Start des Schedulers.

5 Zusammenfassung

Das neue Simulink-Target MB-BOX32 ermöglicht eine strikte Trennung von Applikations-Modell und Basis-Software. Die Einarbeitung für Studierende wird durch die Verwendung der Signal-Schnittstelle stark vereinfacht und beschleunigt. Ohne Modifikationen kann das Applikations-Modell sowohl in der für die Studierenden bekannten Simulink-Umgebung in MiL- und SiL-Tests simuliert werden als auch für die Generierung des Embedded C-Codes verwendet werden.

Für eine weitere Verbreitung der modellgetriebenen Softwareentwicklung für Cortex-M-basierte Steuergeräte in der Lehre ist eine weitere Kostenreduktion der Softwarelizenzen erforderlich. So sollte die Anwendung von MB-BOX32 für Studierende in Verbindung mit MATLAB/Simulink ohne Zusatzkosten ähnlich wie bei Target Support Packages von Mathworks möglich sein.

Literaturverzeichnis

- [BR11] Broy, M., S. Kirstan, H. Krömer, B. Schätz: What is the Benefit of a Model-Based Design of Embedded Software Systems in the Car Industry? In C. B. Jörg Rech, *Emerging Technologies for the Evolution and Maintenance of Software Models* (S. 343-369). IGI Global.
- [MW14] The Mathworks Inc: ARM Cortex-M Support from Embedded Coder. <http://www.mathworks.de/hardware-support/arm-cortex-m.html>
- [ST14] STMicroelectronics: STM32-MAT/TARGET. <http://www.st.com/web/en/catalog/tools/PF258513>.
- [WA14] Aimagin Co Ltd.: Waijung Blockset. <http://waijung.aimagin.com>, Bangkok, Thailand.

Software Design Means for Digital Passenger Cars

Ulrich Freund

FH-Aachen

Fachbereich für Elektrotechnik und Informationstechnik

Eupener Straße 70

52070 Aachen

freund@fh-aachen.de

Abstract: There are indications that the powertrain architecture of the majority of future passenger cars either will be purely electrical vehicles or downsized combustion engine vehicles. Premium cars and sports cars are likely to be plug-in hybrids, using mechanical powertrain components of mass production passenger cars, resulting in all-wheel-driven (AWD) cars with no mechanical torque coupling between front- and rear-axle. As a result, vehicle dynamics of this car category depend so heavily on the applied control software that a new term emerges: The digital passenger car. This paper sketches a design approach based on torque-augmented functional models for executable specification and characteristic map automata for implementation.

1 Introduction

For more than 100 years, the powertrain architecture of passenger cars was determined by the combustion engine. Powertrain- and chassis control electronics, including the software, are just an add-on to the mechanics, replacing control solutions, which were difficult to realize with pure mechanical systems. For example, carburetors were replaced by engine management ECUs that minimize the fuel consumption of combustion engines. The actual requirements for CO₂ reduction results not only in the electrification of the powertrain, but also in efficiency optimization the combustion engine itself.

Compared to the experience in combustion engine vehicles, electric vehicles are still in its infancies. There are many technical- and non-technical issues to solve. Furthermore, electrical vehicles are not appreciated by every customer. This makes it difficult for the automotive industry to develop appropriate vehicles. It appears that the vehicle

manufacturers have two different strategies to cope with this situation for their mass-production cars:

1. One strategy is to provide one vehicle model with either a combustion engine or an electrical motor for propulsion, both models can be manufactured on the same production line. The chassis and the bodywork are, roughly speaking, the same.
2. The other strategy is to develop dedicated vehicles for different propulsion concepts. E.g., there is a combustion engine model line and an electrical vehicle model line. They have different chassis and bodyworks and cannot be manufactured on the same production line. However, they can share components.

Vehicle manufacturers using the first strategy might have to cope with many powertrain variants¹ for one vehicle model:

- Pure electric powertrain
- Electric powertrain with range extender
- Hybrid powertrain
- Plug-In hybrid powertrain
- Pure combustion engine.

In the first strategy, premium cars tend to be hybrid vehicles based on classical combustion engine cars.

The second strategy however introduces a new style of premium vehicles using a downsized combustion engine to drive one axle and the electrical powertrains to drive the other axle. The resulting car is a plug-in hybrid with all-wheel drive (AWD). It has no mechanical torque link between the front- and the rear axle and uses mechanical components of mass production passenger cars. As a result, vehicle dynamics of this car category depend so heavily on the applied control software that a new term emerges: The digital passenger car. A sub-category of passenger cars are sports cars. Some journalists already describe this new generation plug-in all-wheel drive sports cars as digital sports cars [sal14] and classify purely combustion engined vehicles as analog cars. Unfortunately, they do not classify purely electrically driven vehicles yet, maybe because electrical cars are mainly seen as city commuters.

¹ This list is not exhaustive and does not include affine-grained list of different hybrid-powertrain architectures

2 Powertrain Architecture vs. Function Architecture

The mechanical assembly of the drivetrain components for a digital passenger car from powertrain components of an electrically driven and a combustion engine driven analog car is a bottom-up process. The configuration of the control software instead can be seen as a top-down process [Fr12]. This means that the significantly simpler control software for (analog) mass-production passenger car can be seen as variant of the complex control software for digital passenger cars. A functional architecture of a digital passenger car must contain variation points.

2.1 Powertrain & Driving Dynamics

Digital passenger cars are assembled from mass-production powertrain components of an (analog) electrical vehicle and an (analog) combustion engine vehicle. Due to the purely software controlled all-wheel-drive with no mechanical torque link digital passenger cars show driving dynamics superior to their (analog) mass-production counterparts. Even if one considers combustion engine vehicles with computer controlled all-wheel drive systems driving dynamics are influenced by the mechanical coupling between the axles. It is the flexibility, which makes the driving dynamics of digital passenger cars to some extent “freely programmable”.

Driving or vehicle dynamics can be split in longitudinal dynamics, lateral dynamics and vertical dynamics. Longitudinal dynamics deal with the powertrain and the brakes, lateral dynamics with the chassis, the suspension and steering during cornering while vertical dynamics deal with the relationship of the wheels and the body, also dealing with suspension. Needless to say, that tires play an important role in vehicle dynamics too. To assess the quality of vehicle dynamic features, test-scenarios have been designed for decades. Besides acceleration and braking, there are scenarios for lane-changes and skid-pad (Kreisbahn). All kinds of vehicle dynamics interact. If one accelerates in a corner, it depends on the powertrain architecture² whether the car will oversteer (typically rear-wheel-drive) or understeer (typically front-wheel-drive). In digital passenger cars, the dynamic behavior is given completely by the applied control software. Design and calibration of the control software is therefore of paramount importance.

2.2 Functional Architecture

Metamodels for the design of embedded distributed real-time systems like Spes [PH⁺12] or EAST-ADL [ITE08] are structured into perspectives (Spes) or layers (EAST-ADL). The upper layers typically link requirements to a functional model. If the functional model focuses on the system structure, it is called functional architecture. A functional architecture can use the Cartronic structuring concept [WF04]. The building block of

² The behavior of passenger cars with all-wheel drive based on a mechanic torque-link, e.g. Audi Quattro, depends on the mechatronic implementation effort. Sophisticated solutions require electronically controlled differentials and “active”-torque-vectoring which are expensive to manufacture and finally yet importantly quite heavy weighted.

Cartronic is the functionality, a kind of a component. The connectors can be classified to order, request or inquiry. Each class has a dedicated meaning. There is the order, meaning that the actual information will finally applied by an actuator to the plant. A request means that functionality asks another functionality to apply something, e.g. torque, to the vehicle. However, it is not guaranteed that the request will be granted completely. There might be reasons why the other functionality does not grant the request completely. Inquiries provide data, e.g. actual values of sensors or state or output-variables of controllers. A functionality, which has to arbitrate between several requests or limit the request based on inquiry data, is called coordinator. An order of functionality can be computed by arbitrating between different requests and taking into account inquiry data.

In an electronic control system, the flow of torque is represented by signals and is therefore just a flow of data, the energy to apply the momentum to the plant is generated independently in the actuator. In principal, a functional architecture in Cartronic notation acts also as coarse grain data-flow description of a control-algorithm.

The functional architecture for a given vehicle depends on the actuators, generating the momentum. To ensure the vehicles longitudinal and to some extend lateral movements, the actuators are combustion engines, gearboxes and electrical motors, i.e. the components representing the powertrain of a vehicle. With the variation in future powertrain architectures, it is easy to imagine a variety of functional architectures. A functional architecture will look differently for a pure combustion engine driven vehicle, an electrical-motor driven vehicle or a plug-in hybrid. [Dür04] shows how Cartronic structuring models can contain variation points. Cartronic was also used as upper layer structuring mechanism in AutoMoDe [BB⁺05].

3 Variation Points in Engine Management Systems

AUTOSAR classifies variation points according to their definition during development. There are pre-compile-time, link-time and post-build variation points. For post-build, one can distinguish between post-build loadable and post-build selectable.

3.1 Compile Time

At compile-time, variants are built using pre-processor definitions of a compiler. The pre-processor selects the code-fragments relevant for building the system. One needs to maintain at least two files, one file is the code with variation points, and the second file defines the setting of the variations.

3.2 Link Time

At link-time, variants are described by configuration data provided in data-structures, e.g. structs in C. The code using the data can be compiled without the knowledge of the data in the structures. The data however is provided at link time. As a result, the data in

the linked system cannot be changed anymore at runtime. Typically, the configuration data resides in ROM locations.

3.3 Postbuild

At postbuild time, data is provided externally via flashing. Flashing is typically done at the end of a vehicle's production line and requires more expensive flash memory (compared to simple ROM). Post-build loadable means that the data of a parameter or a characteristic map is flashed, while at post-build selectable, two characteristic maps are already programmed in ROM, and only a dedicated parameter is flashed. The value of the dedicated parameter selects the actual map. During the lifetime of a vehicle, the non-selected characteristic map remains unused. However, production-cost of an ECU with several unused maps in ROM might be lower than the production cost of an ECU using a single flashed map instead.

3.4 Postbuild Programming

The postbuild mechanism is not only used for variations in the software, but also for parameter tuning of a control-algorithm. The parameters are set either at a dynamometer or at the test-track. The tuning process, also called calibration³, requires a running ECU. Therefore, the software has to be built first before the calibration can be performed. Quite often, a characteristic map replaces an arithmetic expression.

The calibration process is as important as the software-development itself, hence the same means for version- and configuration management are applied.

Powertrain software for combustion engines depends heavily on characteristic maps. Typically, there is one basic algorithm and all adaptations to the actual engine are done via calibration. During the calibration phase, the data of the characteristic maps is set. Calibration is typically done on a dynamometer. Supplier of engine management ECU⁴s build one "generation" with one μ C family running one algorithm and deliver this system to a variety of engines with different construction characteristics, e.g. in-line or V, number of cylinders, turbo-charged or normally aspirated.

These mechanical construction means of an internal combustion engine are merely reflected by the data of the characteristic maps. The "generation" design approach is also the reason for the slow adaptation of engine management ECUs to the AUTOSAR software architecture. Of course, handling the parameters is a nightmare and requires appropriate tooling and strict development processes.

³ In German language, calibration is translated with Application. This might lead to misunderstandings when talking about software and application.

⁴ ECU stands for Electronic Control Unit, i.e. the embedded system executing real-time software in vehicles.

4 Characteristic Map Automaton

A characteristic map connects input data to output data in an arbitrary manner. In system theory, maps are used when there is a non-linear dependency between input- and output of signals, but one is not able to express this dependency by functions, e.g. polynomials.

The notion of an automaton implies the existence of a state. The actual state of a system expresses its past. The next state is given by the actual state and the input put signal. Output signals depend on either the state or the state and the inputs.

Input signals as well as states might value and time continuous⁵, value discrete and time continuous, value discrete and time discrete. Value and time discrete systems might have a countable but infinite number of patterns and states.

A characteristic map automaton has input signals, state-variables. It generates output signals. The state-map SM defines how the next state will be calculated based on the actual state and the actual input signal. The output-map OM defines how the output is calculated based on the state and the actual input. State-variables hold the actual state.

The hierarchy in **Fehler! Verweisquelle konnte nicht gefunden werden.** shows that the kind of the implemented system depends on the actual data of a CMA's state map and output map. The data itself defines what kind of dynamic system the CMA implements. It can also be seen that the CMA is a low-level construct.

The resulting system is validated during tests on dynamometers and by test-track driving. A system model on a higher level is not reconstructed from actual parameter data.

In engine management systems, one does not find a CMA explicitly. Instead, the control algorithm is a mixture of high-level systems like Mealy automaton or PID-control intermingled with characteristic maps. Typically, a characteristic map combines just two input signals with an output, resulting in a three-dimensional map. Since the visual perception of humans is limited to three dimensions, calibration engineers can cope well with three-dimensional maps. Higher dimensions are typically realized by cascading several three-dimensional maps. Algorithmic parts that are not subject to variation are expressed by arithmetic expressions, automaton or "if-then-else" constructs.

⁵ Continuous means quasi-continuous, i.e. the value discretization means the use of floating point numbers, the time discretization means the use of a numerical solver.

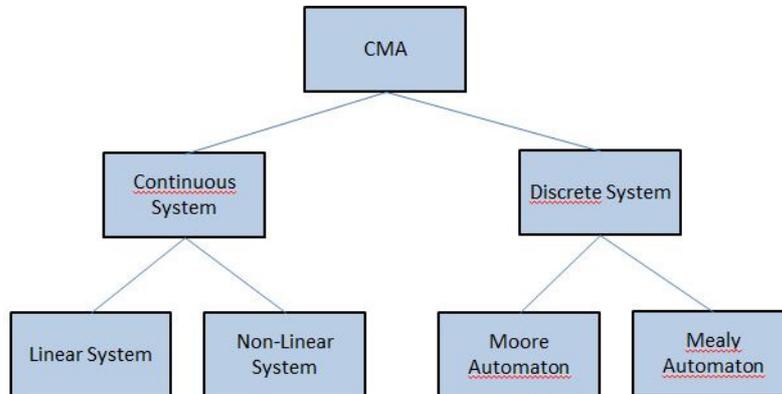


Figure 1: Characteristic Map Automata and its relationship to classical dynamic systems

In chassis-control systems, the number of parameters in the control algorithm is limited. This is because the physics of the driving dynamics can be expressed by differential equations. As a result, control algorithms represent non-linear dynamic systems or automata.

5 Software Design Means for Variable Powertrain Architectures

To cope with the powertrain variability, it proposed to start with the design of a Cartronic functional architecture for a digital passenger car, i.e. an all-wheel-drive plug-in hybrid vehicle. This functional architecture is called the initial functional architecture. The functional architecture contains torque fusion, torque-split and torque transformation functionalities. The functional architecture has to be augmented by

- Quantified torque requirements of the components
- Quantified torque flows between the components
- High-level behavioral description means like finite state machines or differential equations forming non-linear systems for the components.

The augmented functional architecture model is an executable functional model. Appropriate vehicle-models have to be available for driving scenarios. The functional model is simulated in closed loop for the driving scenarios. Torque requirements are adjusted during simulation scenarios.

When all driving scenarios can be fulfilled with the functional model, variants can be created. Creating variants means that a powertrain variant contains fewer components,

e.g. a front-wheel-driven car with combustion engine. All variants will be simulated too and achievable performance figures will be given for the driving scenarios. It is expected that in particular the torque-split and the torque-distribution components will be less complex compared to these components in the initial functional architecture. The transformational components are likely to vary in performance figures. This is because the electrical and mechanical components they control will vary in performance figures for different vehicle models.

If all variants are specified, simulated and adapted, the components can be transformed to AUTOSAR software components. The behavior of the torque-split, -fusion and – transformation components will be represented as internal behavior of an AUTOSAR software component. To account for the powertrain variability, the internal behavior will realize characteristic-map-automata. The high-level behavioral descriptions of the functional model components will be transformed to characteristic data of the CMAs. They will serve as initial data for the post-build programming process.

6 Summary

In future, digital passenger cars will be a composition of powertrain components used in downsized combustion engine vehicles and electric vehicles, thus forming a plug-in hybrid with all-wheel drive. The control software reveals superior driving dynamics of digital passenger cars compared to their (analog) mass-production counterparts, i.e. cars being either pure electrical cars or pure combustion engine cars. In particular, the control software coordinates two powertrain control algorithms and implements lateral control functionality. While the mechanical assembly of the drivetrain components is a bottom-up process, the configuration of the control software is a top-down process. It is proposed to specify the control software of digital passenger by means of torque annotated functional models while the control software is implemented as a network of characteristic map automaton.

References

- [WF04] M. Walther, P. Torres Flores, T. Bertram: CARTRONIC als Ordnungskonzept für den Systemverbund – Analyse mechatronischer Systeme im Kraftfahrzeug, in G. Walliser, Elektronik im Kraftfahrzeugwesen, 4. Auflage, Expert-Verlag, Renningen 2004.
- [Dür04] I. Dürrbaum et al.: Funktionale Modellierung varianter mechatronischer Systeme, Modellierung 2004, Proceedings zur Tagung, 23.-26. März 2004, Marburg.
- [BB*05] Andreas Bauer, Manfred Broy, Jan Romberg, Bernhard Schätz, Peter Braun, Ulrich Freund, Núria Mata, Robert Sandner, and Dirk Ziegenbein. AutoMoDe— Notations, Methods, and Tools for Model-Based Development of Automotive Software. In Proceedings of the SAE 2005 World Congress, volume 1921 of SAE Special Publications, Detroit, MI, April 2005. Society of Automotive Engineers.
- [ITE08] ITEA. The EAST-ADL II, EAST-EEA Website. <http://www.east-eea.net>, January 2008.

- [PH⁺12] Klaus Pohl, Harald Hönninger, Reinhold Achatz und Manfred Broy. Model Based-Engineering of Embedded Systems: The SPES 2020 Methodology Springer, 2012.
- [Fr12] Ulrich Freund: Polymorphic Automotive Control System Modeling beyond AUTOSAR Bosch ETAS - Workshop on Model-Based-Design, Stuttgart, 2012.
- [AR13] The AUTOSAR Layered Software Architecture Release 4.1.2
http://www.autosar.org/download/R4.1/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf
- [sa114] Sport-Auto 01/14, „Vergleich Porsche Carrera GT vs. Porsche 918“, Motor-Presse Stuttgart, 12, 2013

On the right Degree of static Model Analysis for ISO 26262

Dr. Heiko Doerr, Dr. Ingo Stuermer

Model Engineering Solutions GmbH
Friedrichstrasse 55
10117 Berlin
Germany
doerr@model-engineers.com
stuermer@model-engineers.com

Abstract: ISO 26262 has been introduced to capture the state-of-the-art of developing safety-relevant software systems. The standard requires definition and application of modelling guidelines to ensure high quality of models. Still, the standard remains unspecific on the requested properties by intention. Real Projects need therefore to determine the right set of guidelines. A four stage approach derives a reference set of guidelines from the publically available set of guidelines. That reference set by construction meets the requirements of ISO 26262.

1 Motivation: Static model analysis is required by ISO 26262

ISO 26262 has been introduced to capture the state-of-the-art of developing safety-relevant software systems in the automotive domain. Experts gathered and aligned their assessment on the contribution of technologies to the development of safety-related systems. For the different levels of criticality, the experts determined which constructive and analytical measures shall be applied. Implementing the ISO 26262 in the development of safety-related systems will ensure that engineers apply state-of-the-art. Consequently, the required level of quality and safety has been achieved and the risk of liability cases is limited.

ISO 26262-6 requires the application dynamic and static quality measures for traditional and model-based SW development. The complementary set of measures assures the quality of the developed software such that the risk of mal-function of the safety-related software can be reduced. The measures fall into two categories: Static analysis of the SW artifacts addresses mostly non-functional issues whereas the dynamic measures ensure functional correctness by testing.

ISO 26262-6 (see figure 1) lists several topics for which the definition of modeling and coding guidelines shall improve reliability and robustness of models. The topics have been considered as best practice and effective measure to assure the quality of safety-related software systems. So it is advisable for every development project in that area shall apply the guidelines in an appropriate way.

Table 1 — Topics to be covered by modelling and coding guidelines

Topics		ASIL			
		A	B	C	D
1a	Enforcement of low complexity ^a	++	++	++	++
1b	Use of language subsets ^b	++	++	++	++
1c	Enforcement of strong typing ^c	++	++	++	++
1d	Use of defensive implementation techniques	o	+	++	++
1e	Use of established design principles	+	+	+	++
1f	Use of unambiguous graphical representation	+	++	++	++
1g	Use of style guides	+	++	++	++
1h	Use of naming conventions	++	++	++	++

Figure 1: Modelling requirements (from [ISO11])

2 Challenge: How to determine the right set of analyses?

According to the nature of a standard, ISO 26262 focuses on requirements and will not provide specific guidelines to the actual development of safety-related software systems. The topics listed in Table 1 will definitively contribute to quality software, but the definition of concrete modelling and coding guidelines is left open by intention. First, adaptation of new technologies shall be possible. Second, individual projects will apply different processes and tool chains.

Implementation of the standard may be supported by company standards which will tailor the requirements of the standard to the company settings of processes, tool chains and domains. These refinements will capture accumulated knowledge on the implementation of ISO 26262 and will help to implement the standard in the company setting. However, as company standards usually remain at a higher level, they again provide no answer to the right set of static analyses.

Each development project has to determine the appropriate set of measures to meet the requirements of ISO 26262. The solution space for that selection of implementation guidelines has a wide range: simple structural properties like naming conventions at the lower end can be evaluated by simple string operations whereas data flow or timing analyses are at the upper end of mathematical complexity. Without proper guidance, the individual project faces a high risk that the set of selected measures is not appropriate: Either it is not sufficient, and the safety plan may not be accepted by a reviewing instance; or too much effort and consequently time is spent to conduct analyses actually not contributing to the safety of the final software system.

3 Solution: Comprehensive survey and assignment of state-of-the-art analyses

We provide a structured approach to the selection of guidelines for proper implementation of the ISO 26262. The procedure takes up the state-of-the-art argument of standardization and applies it to the selection of proper guidelines. Our approach has 4 stages. The result is an efficient quality assurance plan. The plan will contain the execution of static analyses for those guidelines being automatically assessable. In addition, model reviews will be requested for those guidelines for which no automatic check can be implemented.

We will demonstrate the approach for a specific implementation technology: model-based implementation of software using the Simulink / TargetLink tool chain. That well-established implementation approach is widely used in automotive software development. The approach shown will still be applicable to other implementation technologies or tool chains.

3.1 Stage 1: Baselineing

Our approach extends the liability argument to the selection of guidelines: if, in case of an erroneous functionality, the engineering party can prove that state-of-art has been applied in development, than it is quite hard to claim that the fault is still in the responsibility of the engineering party. We apply that argument to the selection of appropriate modeling guidelines. At first, we have to determine state-of-the-art for modeling guidelines. That state-of-the-art usually is published by professional associations or other organizations which collect best practices and standards. Additional proprietary collections of OEMs or Tier 1s may contain further guidelines. For two reasons, they are not considered as baseline: 1) the proprietary guidelines are usually not available for the public; additionally and 2) these guidelines will cover implementation strategies being local to the development process at the OEM / Tier 1 and will not fit to other companies.

For model-based software development, we find quite a number of engineering practices. The comprehensive list of published guidelines is the following: Mathworks Automotive Advisory Board ([MA12]), MISRA Autocode TargetLink ([MI09]), MISRA Autocode Simulink / Stateflow ([MI07]), dSPACE TargetLink Known Problems, and dSPACE TargetLink guideline documents ([dSP10]). These 5 documents provide around 700 individual guidelines. If all are applied within a development project, state-of-the-art will be used and the requirements of the ISO 26262 standard satisfied according to the state-of-the-art argument.

However, the guideline sets exhibit redundancy so to reduce redundancy, the next stage is entered.

3.2 Stage 2: Consolidation

Due to historical reasons, the published standard documents have individual strengths. Hence all of them need to be reflected when considering the compliance to the ISO 26262 requirements. On the other hand, the detailed assessment of all guidelines will show how individual guidelines may be replaced. That task is an expert assessment because the impact of the individual guidelines must be fully understood before a specific guideline may take over the role of another one. Quite likely, additional constraints or parameterization of the guideline will help to align the baseline set of state-of-the-art.

In the example of guidelines for the Simulink tool chain, the consolidation of guidelines has revealed that nearly 30% of the guidelines were superfluous as they have been duplicates. The consolidation therefore contributes to a significant extend to the reduction of effort for the implementation of static analyses for model-based development.

3.3 Stage 3: ASIL selection

At the start of stage 3, the baseline set of guidelines has been consolidated to a reduced set of guidelines which actually contribute to the implementation of ISO 26262. Stage 3 now continues to further determine the adequate subset of guidelines. Now, the impact of the Automotive Safety Integrity Level (ASIL) level to the quality requirements is taken into account. Each of the guideline now shall be evaluated. Proper combinations of guidelines shall be determined which will e.g. contribute to the “1f) Use of unambiguous graphical notation” being highly recommended for software being used at ASIL B. A subset of the selected guidelines will be sufficient for ASIL A software systems. Based on the ASIL impact analysis, the efficiency of guideline checks will be further improved, since the check set are designed for individual ASIL levels.

3.4 Stage 4: Scheduling

ISO 26262 exhibits one single phase for the mere implementation of software: phase 6-8 “Software unit design and implementation”. Model-based software development usually is further detailed and distinguishes at least function model (on the basis of floating point arithmetic) and implementation model (including the discretization and scaling of signals to the processor platform). Taking this distinction into account is the motivation for further efficiency gains. For each of the guideline the most appropriate phase(s) of development can be determined at which the guideline check shall be applied.

It must be ensured that by frontloading, quality shortcomings are detected as early as possible. For instance, when comparing function and implementation models it is quite reasonable to check for proper layout already at the early development phase of functional model because discretization for implementation does not alter the model that much.

The table below shows the structural principle of the result of the overall assessment. At first, the table lists all guidelines being defined in the baseline of state-of-the-art documents. As the result of stage 2: Consolidation, the table indicates whether a guideline has actually been chosen or whether it can be replaced by comparable guideline. The next area shows the relevance for certain ASIL levels. Information on the right development phase concludes the table.

A sample result is shown in table 1. Here we have a guideline “ma_32”, which has been selected for assessment of ISO compliance. Still, due to complexity reasons, it is not required for lower ASIL. For sake of modelling efficiency, is it appropriate to check the adherence to the guideline already at the stage of functional modelling. So, the developer should not wait to improve the model at the implementation stage.

Table 1: Sample assignment of guideline

Guideline	Choice	ASIL A	ASIL B	ASIL C	ASIL D	Function	Implementation
ma_32	Yes	-	-	x	x	x	-

Summary: Quality assurance plan for model analysis

The staged assessment of modeling guidelines has been applied and validated during the introduction of model-based development by Siemens ([BJ13]). Optimized application of modelling guidelines serves since then for ISO compliance. The approach determines a comprehensive solution which is fully implementing the requirements by ISO 26262. The comprehensive selection of guidelines reduces the number of guidelines still maintaining full compliancy. The approach is specialized for a dedicated tool chain and explicitly collects a full baseline of guideline documents. Therefore, the resulting table can be incorporated into an overall quality assurance plan addressing the area of model analysis.

References

- [ISO11] Road vehicles - Functional safety - Part 6: Product development: software level, ISO 26262-6:2011(E), 2011
- [MA12] Control Algorithm Modeling Guidelines using MATLAB®, Simulink®, and Stateflow®, V3.0, MAAB, 2012
- [MI07] MISRA AC TL Modelling style guidelines for the application of TargetLink in the context of automatic code generation, V1.0, MIRA, 2007
- [MI09] MISRA AC SL SF Modeling design and style guidelines for the application of Simulink and Stateflow, V1.0, MIRA, 2009
- [dSP10] Modeling Guidelines for MATLAB/Simulink/Stateflow and TargetLink, V3.0, dSPACE, 2010
- [BJ13] Process for Functional Safety, Brothank, Jung et al. dSPACE Magazine 1/2013, 2013

Effort and Efficacy of Tool Classification and Qualification

Mirko Conrad, Ines Fey

samoconsult GmbH
Berlin, Germany

{mirko.conrad|ines.fey}@samoconsult.de

Abstract: With the advent of ISO 26262 in 2011, a topic that was previously only of interest to a small group of specialists, has become an obligation for automotive OEMs, their suppliers, and for software tool vendors: tool classification and qualification according to ISO 26262-8. Since automotive development organizations may utilize up to a few hundred tools, conducting tool classification and qualification requires significant efforts.

To aid planning the required activities to gain confidence in the software tools used, this paper will provide an effort estimate for tool classification and qualification according to ISO 26262. The effort estimate is based on data points from practitioners in the field as well from available literature. While many of the data points used stem from model-based development projects / tools, the authors believe that the results are applicable to not limited to this software engineering paradigm.

Spending significant efforts on gaining and increasing confidence in the use of software tools also raises the question on the efficacy of the corresponding activities. Therefore, the effort estimate will be augmented with available data on the efficacy of tool qualification.

1 Tool classification and qualification according to ISO 26262

Tools can potentially contribute to safety by automating activities they perform and predictably performing functions that may be prone to human error. On the other hand, tool errors may adversely affect system functionality or safety if a tool inadequately performs its intended functions [IP57].

Therefore, recent functional safety standards typically require dedicated activities to gain confidence in the tools used. These activities are commonly referred to as tool qualification activities. ISO 26262 [ISO26262], the functional safety standard for automotive E/E systems, is no exception.

1.1 Overview of the ISO 26262 approach

ISO 26262-8, clause 11 defines the necessary process to gain confidence in the usage of software tools. It outlines a two step process consisting of:

- 1) a tool classification to determine the required level of confidence in a software tool.
- 2) a formal tool qualification to establish the required confidence.

While the tool classification is a mandatory step, the need for tool qualification depends on the result of the tool classification step. Tool qualification is only required if the required confidence exceeds a certain threshold.

Please note, that the ISO 26262 tool classification / qualification approach does not distinguish between different tool categories such as development tools and verification tools.

The approach is illustrated in Fig. 1 and will be summarized in the following two subsections. More detailed discussions can be found e.g. in [Con10, CSM11, HKW+11]. Tool classification and / or qualification examples incl. a model-based verification tool can be found in [Mai09, CSM10, CF11]

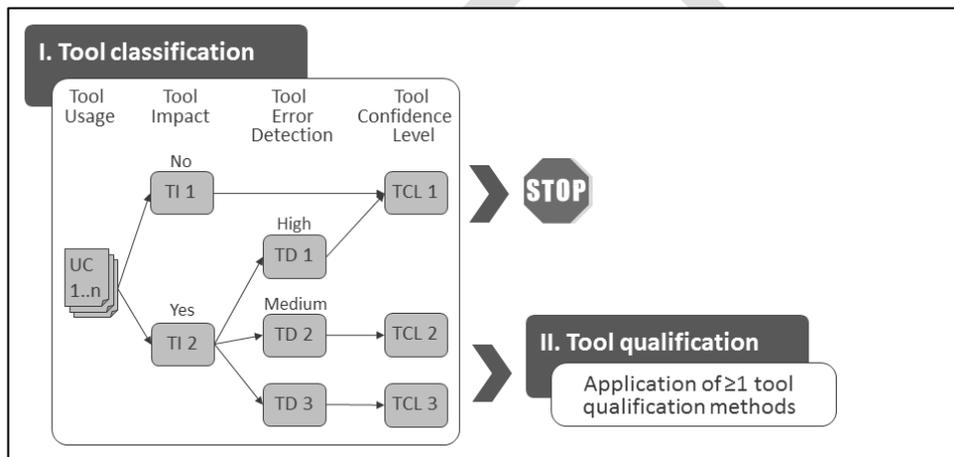


Figure 1. Two step approach to gain confidence in the software tools used

1.2 Tool classification

In step I, the intended usage of the tool, i.e. the tool use case(s), needs to be documented, analyzed, and evaluated to ascertain

- the possibility that a malfunction in the software tool can introduce or fail to detect errors in the system being developed. The result is expressed using one of two tool impact categories (TI 1 or TI 2).
- the confidence in measures to prevent or detect a malfunctioning tool and corresponding erroneous output. The result is expressed using one of three tool error detection classes (TD 1, TD 2 or TD 3).

As a result of this analysis, a required tool confidence level is determined. The tool confidence level is classified using one of three tool confidence levels TCL 1, TCL 2, or TCL 3.

1.3 Tool qualification

Tools with the lowest possible TCL (i.e., TCL 1) do not require subsequent tool qualification. For all other TCLs, formalized tool qualification is necessary. The selection of appropriate tool qualification methods depends on the required TCL and on the Automotive Safety Integrity Level (ASIL) of the safety-related system to be developed using the software tool.

ISO 26262-8 recognizes the following tool qualification methods:

- a) Increased confidence from use
- b) Evaluation of the tool development process
- c) Validation of the software tool
- d) Development in compliance with a safety standard

Fig. 2 shows the degree of recommendation of these four methods for the various ASILs and TCLs. In the figure, the recommendation level of a tool qualification method for a given ASIL and TCL is indicated by the height of the corresponding column. ‘+’ refers to ‘recommended’ method, ‘++’ to a highly recommended method.

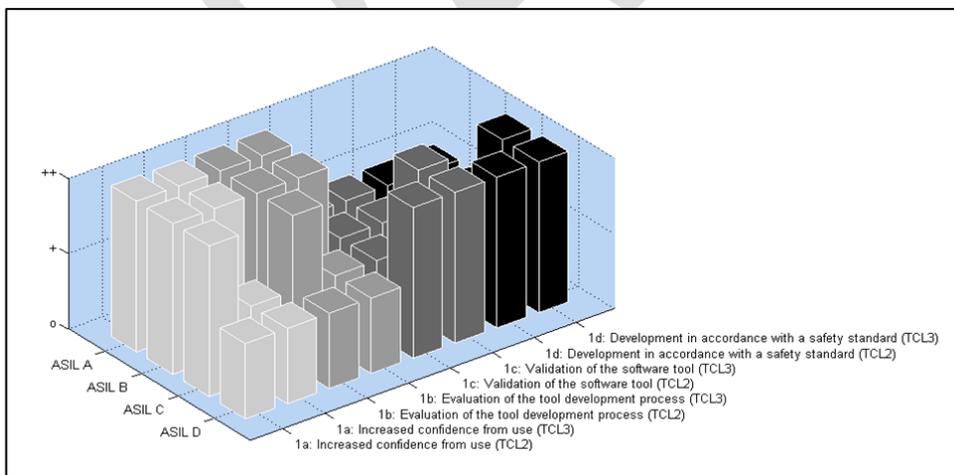


Figure 2.

Recommendation levels for the ISO 26262 tool qualification methods

None of the tool qualification methods is highly recommended for all possible ASIL – TCL combinations, i.e. there is no ‘preferred’ tool qualification method that fits all situations. To qualify a tool for all ASILs, it might be expedient to utilize a combination of multiple tool qualification methods.

2. Tool classification and qualification effort

2.1 Preliminary remarks

Since the tool classification step is a mandatory step in the process and the tool qualification is only necessary for tools with a resulting maximum TCL of two or three, one would expect to find the lowest total efforts for TCL 1 tools and the highest total efforts for TCL 2 or 3 tools, i.e. tools that require an actual tool qualification on top of the tool classification. Furthermore, an initial classification / qualification of a tool is typically more costly than a re-classification / re-qualification of a subsequent version of the same tool.

2.2 Extreme cases

As a starting point for providing a realistic range for tool classification / qualification efforts, we would give a lower and an upper bound based on two extreme cases the authors came across.

The least elaborate ISO 26262 tool classifications the authors came across, comprise one row in an Excel classification spreadsheet per tool. We would estimate less than 15 mins to create such an entry. Use cases are typical very coarse and manual reviews or references to some existing tool qualification kits are frequently mentioned as error mitigations methods.

The most elaborate tool classification / qualification in the context of ISO 26262 the authors are aware of, is probably a validation suite for an autocode tool chain comprising of the TargetLink code generator, a WindRiver C cross compiler and a cross linker [SML08]¹. According to [SML08], the test suite which is based on tool qualification method c, i.e. validation of the software tool, comprises 6730 test models and more than 150 million test points (i.e. comparisons between actual and expected values). The effort to create such a test suite very likely exceeds 1 person year².

These two examples are just provided as extreme cases. It’s out of scope of this paper to assess the standard compliance or the effectiveness of these approaches. However, the effort of tool qualifications in practice is likely bounded by those extremes.

¹ also known as TVS or VASE

² Please note that this is an effort estimate for the initial tool qualification using the validation suite. Subsequent re-qualifications for different code generator / compiler / linker versions are significantly less costly since they mainly re-use existing test models and test points. The manual effort for a re-qualification may be in the range of a few person weeks.

2.3 Estimation approach

To get a more typical estimate, the authors combined data obtained from industry experts at Kugler Maag CIE, Validas AG, and samoconsult GmbH September 2013.

Where possible / useful, the data was augmented or aligned with published data. Statistical outliers, like the ones presented in section 2.2 were not taken into account.

Each of the three companies provided minimal, maximal, and typical tool classification and tool qualification efforts as well as minimal, maximal, and typical tool chain sizes.

The per tool effort estimates and the tool chain size estimate were determined by calculating the arithmetic mean of the three minimal efforts/sizes, the three maximal efforts/sizes, and the three typical efforts/sizes respectively.

A preliminary version of the results was presented in [CF13].

2.4 Classification and qualification effort per tool

Following the approach outlined in subsection 2.3 resulted in a mean minimum tool classification effort of 2 person hours and a mean maximum tool classification effort of 53 person hours per tool. The average typical classification effort was 14 person hours per tool.

The mean minimum tool qualification effort was 21 person hours, the mean maximum tool qualification effort 300 person hours, and the mean typical qualification effort 80 person hours per tool.

Because only a subset of the tools which are classified need to be qualified down the road, the data set used for the qualification estimate was smaller than the one used for the classification estimate.

Fig. 3 summarizes the per tool effort estimates.

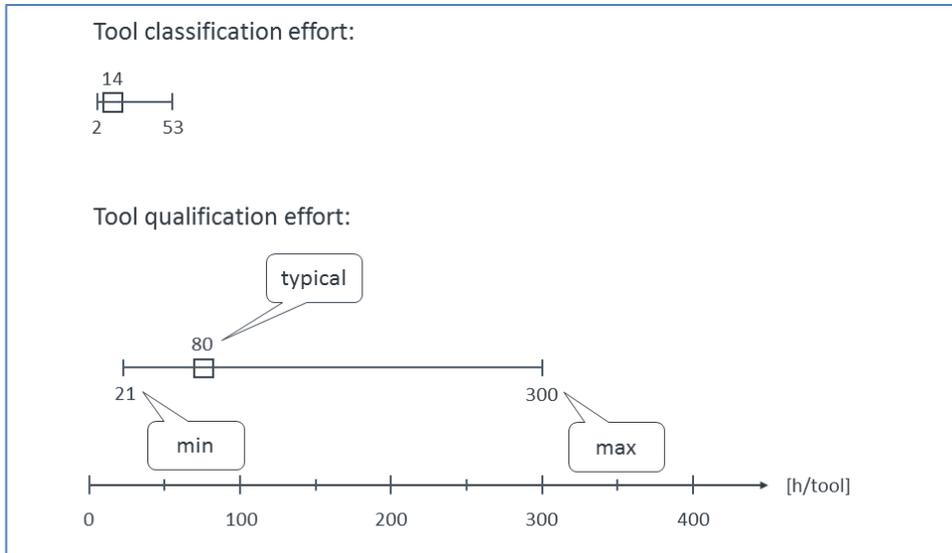


Figure 3. Minimal, typical, and maximal duration of ISO 26262 tool classification and qualification activities

2.5 Tool chain size and TCL distribution

To estimate the overall tool classification and qualification effort per organization, we also needed information on the tool chain size.

Published data from [HKW+11] suggests that the up to 1500 tools might be utilized at a large automotive organization, taking different business units into account. However, the authors considered this an extreme case as well and did not incorporate this number as a data point for the estimate. Rather, the tool chain size was also estimated based on the approach outlined in subsection 2.3.

The average minimum tool chain size was 12 tools, the average maximum tool chain size 50 tools, and the average typical tool chain size 50 tools.

An OEM might use a smaller number of tools than a supplier, especially if the OEM only covers a subset of the development life cycle (e.g. the ‘upper’ parts of the V model).

Tool chain sizes may also vary depending on whether an organization counts different versions of the same tool as one tool or each version as a separate tool. Since each tool version needs to be classified and potentially qualified, an effort estimate should account for different tool versions. However, the classification / qualification effort for the first tool version is typically much higher than for subsequent versions.

The number of tools to be classified equals the tool chain size. To determine the number of tools to be qualified we need to gauge the fraction of TCL 2 and TCL3 tools of the tool chain.

[HKW+11] provides such a data point for the TCL distribution of a tool chain. The authors report that 98% of the tools were classified as TCL1, whereas only 2% of the tools had a maximum TCL of 2 or 3 and needed to be qualified.

In practice, the portion of TCL 1 tools may differ depending on whether an organization is ‘process heavy’ or not. Process heavy organizations have more checks and balances to detect potential tool anomalies. As a result, the TCLs for the individual tools may be lower than in other less process oriented companies.

From their own experience, the authors would estimate 5% of the tools to be TCL 2 or 3 tools. This value was used in our estimate to determine the number of tools in the tool chain that need to be qualified (non integer results were rounded up).

This results in 1 tool to be qualified for the minimum tool chain size, and 20 tools to be qualified for the maximum tool chain size. In a tool chain of typical size, 3 tools would need to be qualified.

The number of tools to be classified and qualified is illustrated in Fig. 4.

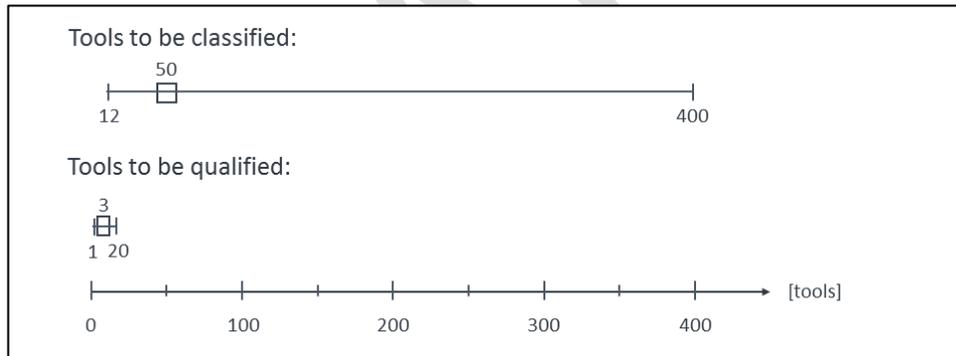


Figure 4. Minimal, typical, and maximal number of tools to be classified and qualified

2.6 Overall tool classification and qualification effort

Using the data from subsections 2.4 and 2.5, the tool classification and qualification efforts for the overall tool chain can be estimated.

Assuming a typical tool chain size as per subsection 2.5 and typical classification and qualification efforts per tool as per subsection 2.4, the tool classification effort is 700 person hours and the tool qualification effort is 240 person hours. The corresponding minimal and maximal efforts are summarized in Fig. 5.

Although the per tool qualification effort is larger than the per tool classification effort, the total classification effort outweighs the total qualification effort.

Tool Classification					
		effort [h]			
		min	typical	max	
# tools	min	12	24	168	636
	typical	50	100	700	2.650
	max	400	800	5.600	21.200

Tool Qualification					
		effort [h]			
		min	typical	max	
# tools	min	1	21	80	300
	typical	3	63	240	900
	max	20	420	1.600	6.000

Figure 5. Estimation of the ISO 26262 tool classification and qualification effort for the entire tool chain

The combined overall effort for the ISO 26262 tool classification and qualification activities in a typical case is 940 person hours or 23.5 person weeks. Assuming an hourly cost of 100€, tool classification and qualification would cost about 94.000€.

The tool classification / qualification effort might be seen as a hindrance when choosing to use software tools. Unfortunately, there seems to be no ISO 26262 specific data to verify or falsify such a hypothesis.

However, there is related data from the civil avionics development community stemming from a survey conducted as part of the Federal Aviation Administration's (FAA) commissioned Streamlining Software Aspects of Certification (SSAC) program [HDK+99]. This survey included a section on tool qualification. The results of this survey indicated that 60% of the respondents considered the cost attributed to tool qualification to be small or negligible, 36% considered the cost to be substantial, and only 4% considered the cost to be prohibitive [HDK+99].

3 Tool qualification efficacy

One of the concerns regarding tool qualification is whether the overall quality of the system under development will benefit from conducting tool qualification activities. In particular, people frequently doubt whether the tool qualification process can find tool errors.

Here, the data from the above mentioned survey [HDK+99] speaks a clear language: The survey results indicate that errors had been found in tools during the qualification process. Out of the survey respondents with tool qualification experience, ~44% found errors with a development tool and ~57% found an error with a verification tool during tool qualification (cf. **Error! Reference source not found.**Fig. 6)**Error! Reference source not found.**

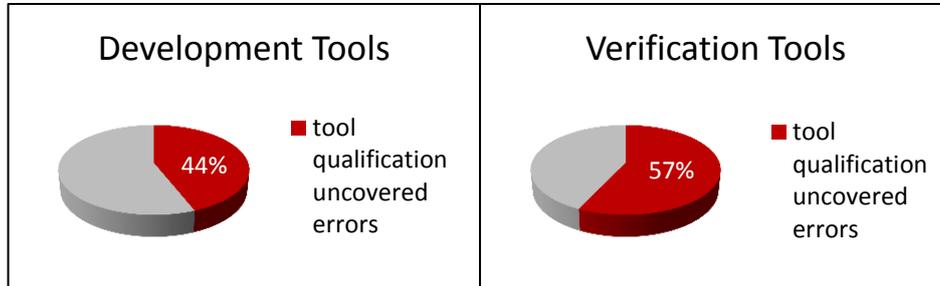


Figure 6.

Tool qualification efficacy [HDK+99]

Another data point for the efficacy of tool qualification activities is [SLM08]. The authors report that an instance of their validation suite for a model-based code generator / compiler / linker tool chain detected “30 distinct code generation issues”. This validation suite was developed in the context of a draft version of ISO 26262, but covers requirements from other sources as well.

3. Summary and conclusion

Given the limited published data on the effort of the ISO 26262 tool classification and qualification activities, the authors estimated these efforts for realistic cases.

As a result, a typical tool classification effort is estimated to be 1.75 person days per tool, a typical tool qualification effort 10 person days per tool. The total tool classification and qualification effort for a tool chain of typical size is estimated to be 23.5 person weeks.

The estimates are based on data points provided by three companies that provide tool classification and qualification services to the automotive development community. Although the approach to estimate the necessary efforts is pragmatic in nature, the authors believe that the resulting numbers are in the right ball park and might be helpful to automotive organizations that want or need to estimate their tool qualification efforts.

The authors would like to encourage others who have additional data points to share those with the authors in order to broaden the data set and to improve the estimates.

The estimates for the ISO 26262 classification and qualification efforts were augmented with available data on the efficacy of tool qualification activities.

Acknowledgements

The authors would like to thank Oliver Slotosch (Validas AG) and Bonfiaz Maag (Kugler Maag CIE) for providing effort figures from their respective companies. These estimates were an important prerequisite to come up with a realistic effort estimate for the automotive community.

References

- [BB09] M. Beine, A. Bärwald: Einsatz zertifizierter Codegenerierungswerkzeuge in sicherheitsgerichteten Entwicklungen. Safetronic 2009, Munich, Germany, 2009.
- [CF11] M. Conrad, I. Fey: ISO 26262 - Exemplary tool classification of Model-Based Design tools. Softwaretechnik-Trends Band 31 (2011) 3
http://pi.informatik.uni-siegen.de/stt/31_3/01_Fachgruppenberichte/ada/5-CF11-11_20110803.pdf
- [CF13] M. Conrad, I. Fey: Quo vadis tool qualification. 5th EUROFORUM Annual Conference 'ISO 26262', Stuttgart, Germany, Sept. 2013
- [CMR10] M. Conrad, P. Munier, F. Rauch: Qualifying Software Tools According to ISO 26262. Proc. of MBEES 2010, Dagstuhl, Germany, 2010
- [Con10] M. Conrad: Software Tool Qualification According to ISO 26262 - An Experience Report. Supplementary Proc. of 21. Int. Symposium on Software Reliability Engineering (ISSRE 2010), pp. 460-466
- [CSM10] M. Conrad, J. Sauler, P. Munier: Experience Report: Two-stage Qualification of Software Tools. 2. EUROFORUM ISO 26262 Conference, Stuttgart, Germany, September 2010
- [CSM11] M. Conrad, G. Sandmann, P. Munier: Software Tool Qualification According to ISO 26262. SAE 2011 World Congress, Detroit, MI, US, April 2011, doi:10.4271/2011-01-1005
- [Glö08] T. Glötzner: IEC 61508 Certification of a Code Generator, ICSS2008
- [HDK+99] K. J. Hayhurst, C. A. Dorsey, J. C. Knight, N. G. Leveson, G. F. McCormick: Streamlining Software Aspects of Certification: Report on the SSAC Survey. NASA/TM-1999-209519, Aug. 1999.
- [HHD+98] K. J. Hayhurst, C. M. Holloway, C. A. Dorsey, J. C. Knight, N. G. Leveson, G. F. McCormick, J. C. Yang: Streamlining Software Aspects of Certification: Technical Team Report on the First Industry Workshop. NASA/TM-1998-207648, Apr. 1998.
- [HKW+11] R. Hamann, S. Kriso, K. Williams, J. Klarmann, J. Sauler: ISO 26262 Release Just Ahead: Remaining problems and Proposals for Solutions. SAE 2011 World Congress, Detroit, MI, US, April 2011.
- [IP57] SC-205 / WG-71 Information Paper IP57 'Tool Qualification Supplement', 2009
- [ISO26262] ISO 26262. International Standard 'Road vehicles - Functional safety'. 2011-2012.
- [KKG10] J. Klarmann, S. Kriso, M. Gebhardt: Qualification of development tools as per ISO 26262. REAL TIMES, 1/2010, pp. 28-20
- [Mai09] M. Maihöfer: Umgang mit Entwicklungswerkzeugen in Software-Entwicklungsprozessen der Automobilindustrie - ISO DIS 26262, Band 8, Kapitel 11: Inhalt, Bewertung, Auswirkung und Umsetzung (in German). EUROFORUM Konferenz 'Funktionale Sicherheit nach ISO/DIS 26262', Stuttgart, Germany, September 2009

- [SML08] S.-A. Schneider, P. R. Mai, T. Lovric: The Validation Suite Approach to Safety Qualification of Tools. Automotive Safety and Security 2008, Stuttgart, Germany, Nov. 2008.

DRAFT

Model-based Parallelization and Optimization of an Industrial Control Code

Ralf Jahr, Martin Frieb, Mike Gerdes, Theo Ungerer

Department of Computer Science
University of Augsburg
86135 Augsburg
{jahr, martin.frieb, gerdes, ungerer}@informatik.uni-augsburg.de

Abstract: With the rise of multi- and many-core processors in industrial embedded applications companies face the challenge of changing legacy single-core applications towards multi-threaded programs. A model-based parallelization approach can be applied like our recently introduced *pattern-supported parallelization approach* with focus on parallel design patterns and hence structured parallelism.

To demonstrate its applicability and the included model-based optimization, a state-of-the-art industrial control code is parallelized. It consists mainly of several periodic tasks and a control loop, which is in the focus. Applying our parallelization approach reveals 11 instances of the parallel design pattern *Task Parallelism*. The assignment of cores to these pattern instances is optimized and the resulting execution time is approximated. This leads to a maximum speedup of 5.7 for 17 cores with parallelization overheads (8.4 with 10 cores without overheads).

1 Introduction

Embedded systems feature more and more frequently multi- and many-core processors. Applications which shall benefit from these additional computation resources need to be programmed concurrently with multiple threads [Sut05].

For the transition from single core software to parallel software, we presented a parallelization approach based on parallel design patterns (PDPs) at MBEES 2013 [JGU13a]. PDPs are abstract concepts (no source code) describing recurring situations of parallelism. In this approach, first a model is created by analysis of the legacy source code and PDPs are placed revealing a high degree of parallelism. Second, this model is optimized for optimal parallelism. As next step after this optimization, one of the found Pareto-optimal configurations has to be selected and the original source code has to be changed correspondingly. Algorithmic skeletons can facilitate the implementation of PDPs.

This paper gives insight into the model-based process of parallelization and optimization of a real-world industrial code, i.e., the control code of a modern construction machine as a prototypical example.

In this case, the main drivers for execution on a multi-core are the need for higher computation power to implement additional and more sophisticated safety features as well as faster execution of algorithms leading to shorter reaction times to external events and still high-precision results. The execution of periodic tasks on dedicated cores and running control loops without intermissions also leads to reduced jitter and higher sample rates.

The main goal of this paper is to show the applicability of both model-based parallelization and optimization for a real-world application. The presented results are transferable and beneficial for similar parallelization projects of legacy single-core control codes.

The remainder is structured as follows: Related work is presented in Section 2. The structure of the control application and parallelization concepts are described in Section 3. The parallelization approach is applied to the main control loop of the application in Section 4 and Section 5 concludes the paper.

2 Related Work

Industrial embedded applications often have code bases which have grown and been improved over many years. For increasing the level of parallelism in such codes, applying a parallelization approach might be more beneficial compared to a re-implementation. The main reason is a big chance for reuse of existing code leading to lower implementation and testing efforts.

The model-based *pattern-supported parallelization approach*, which is applied in this paper, was introduced in [JGU13b] and its applicability for hard real-time embedded systems is discussed in [JGU13a]. The approach introduces parallelism only by PDPs, which present best-practice solutions for recurring situations of parallelism (cf. [KMMS10]). PDPs, which are a theoretic “textual” concept, can be implemented by algorithmic skeletons [Col04] or with a framework like MTAPI [GL13]. This simplifies implementation because already tested code is applied.

Competing parallelization approaches were presented by Mattson et al. [MSM04] and Foster [Fos95]. Both are model-based; Mattson et al. also allow parallel design patterns only for parallelism, Foster defines the process more formally. Our parallelization approach and those by Mattson and Foster have to be done manually or semi-automatically with tool-support.

Instead of a parallelization approach, automatic parallelization could also be applied. Cordes et al. [CMM10, CM12], for example, developed an approach with a *hierarchical task graph* as model which can be optimized automatically, too. This leads to unstructured parallelism making later timing analysis much harder; structured parallelism is very beneficial for this (cf. [JGU13a], [CKW⁺13], and [ORS13]).

Gerdes et al. [GWG⁺11] modified the control application of a large drilling machine manufactured by Bauer Maschinen GmbH for a timing-analyzable processor with two or four cores. A maximum observed execution time (MOET) speedup of 2.62 and a worst case execution time (WCET) speedup of 1.93 were reached on four cores; RapiTime [Sys] was

used for the measurement-based WCET analysis. However, the parallelization was based on knowledge of the application (i.e., domain knowledge), whereas the approach followed in this paper is rather systematic and model-based. Also the target was only four cores in contrast to the about 16 cores targeted in the following.

Comparable to our application are for example automotive electronic control unit (ECU) applications. Software support for model-based development is very strong in this domain. The development of parallel applications is right now starting, the main standard for the software development in the automotive domain AUTOSAR included first concepts for multi-core processors in Version 4.1 [AUT13].

3 Software Structure and Parallelization Concept

In the examined machine, an embedded real-time-capable electronic control unit (ECU) is installed. The software on the ECU, which is programmed in C, comprises three layers: At the lowest level is the so-called **BIOS** (closed-source by the ECU manufacturer) for basic input- and output-operations and which provides a scheduler executing tasks at different priorities and frequencies. These tasks are implemented in other layers of the software and, e.g., send and receive CAN messages, read input values from sensors and set output values to actors. The **middleware** provides functionalities abstracting from the BIOS. It implements drivers for the interfaces and sensors, such as keyboards and joysticks, but also inclinometers for mast orientation, for example. The actual control of the machine takes place in the **application program** which is written specifically for one type of machine. APIs are defined between the different layers and data is exchanged by shared memory.

The code is control intensive (and not data intensive); there are no large loops but many conditions on parameters like the series of the machine, configuration parameters, or sensor values. Many state machines can be found.

The control program executes in two phases: The **initialization** is run once after power-up. The **main control loop** (short: main loop) is executed concurrently to the scheduler invoking **periodic tasks**. Hence the scheduler interrupts the main loop from time to time.

For the execution of the control code on a multi- or even many-core processor, the focus of all effort is put on the main loop and the periodically executed tasks. The initialization is executed only once for a few seconds, hence it can be kept as it is and executed on a single core. In the single-core version, several periodic tasks were not invoked by the scheduler but in the main loop by observing the system time. These tasks were isolated and are also executed by the scheduler in the parallel version.

For all periodic tasks, the concept for parallelization is to run all of them on a dedicated number of cores. Static cyclic scheduling [ATB93] can be used for this; the application program or other periodic tasks are not interrupted anymore. This reduces jitter and leads to higher sample rates. With a look on the maximum observed execution times (MOETs) of the periodic tasks in the single-core system we dedicate two cores of the many-core processor to them. The main loop is parallelized with PDPs as described in the following section.

4 Model-based Parallelization of the Main Loop

Parallelism in the main loop of the control program is extracted (Section 4.1) and optimized (Section 4.2) according to our previous publications on the pattern-supported parallelization approach [JGU13b, JGU13a]. The approach describes a systematic way starting with a sequential and resulting in a parallel program. PDPs are the only allowed means for introducing parallelism, hence the resulting program features structured parallelism.

The applied parallelization approach is model-based. It requires manual work to isolate and describe situations suitable for parallelization. They must match PDPs as described in a Pattern Catalogue; we selected the parMERASA Catalogue [GJU13] with time-predictable design patterns. For modeling the *Activity and Pattern Diagram (APD)* [JGU13b] is used¹.

The following Section 4.1 describes creation of a model of the software and parallelization towards a high degree of parallelism. This parallelism is optimized with the goal of a high speedup on the available cores in Section 4.2. An approximation of the implementation effort in terms of shared variables which have to be synchronized is shown in Section 4.3.

4.1 Revealing Parallelism

The goal of the first phase of the parallelization approach is to construct a model of the software by APDs and to reveal a high degree of parallelism in this model by PDPs. This parallelism should be about a magnitude higher than the level which seems reasonable for the target platform. Both (a) situations for parallel execution fitting design patterns have to be spotted and, what proved to be the more work intensive task, (b) dependencies between the different code parts have to be identified and described. This is necessary to be able to decide if a parallel execution is indeed possible.

The PDPs *Task Parallelism* and *Periodic Task Parallelism* from the selected Pattern Catalogue [GJU13] were found in the control loop. *Task Parallelism* describes the parallel execution of sub-tasks from a joint starting point and finishing with a barrier; after completion of all sub-tasks control is returned to a main thread. *Periodic Task Parallelism* matches to the parallel and periodical execution of tasks as realized by a scheduler.

For identifying situations for PDPs, the source code has to be checked manually. Unfortunately, there is yet no supporting program available for this. Even if available, automatic tools should be used carefully: The main difference between automatic parallelization and our approach is that also situations fitting only roughly (and maybe requiring manual adaptation of the source code) can be identified by the developer whereas automatic tools will only find exact matches.

For each potentially parallel situation it has to be decided if the code fragments – in the model they map to activities – can actually be executed in parallel without any difficulties.

¹The APD is a slightly extended *UML2 Activity Diagram*. The fork/join operator must not be used, instead a new node type similar to an activity is introduced representing an instance of a PDP.

This is mainly dependent on the data and sometimes control dependencies. As first step, for each activity the accesses to shared variables have to be identified as well as the access types (read or write). The second step is to compare the access sets of activities to be executed in parallel: If they are disjoint or if they only match for reading accesses then a parallel execution is very probably possible. If the sets are not disjoint and one or more accesses of their intersections are writing accesses, then a more detailed examination is necessary, also it is probably important to preserve the order in which read and write accesses are performed.

Because of the code size and the number of shared resources it is not possible to annotate the APDs as suggested in the original publication [JGU13b]. Therefore, it was decided to create a list for each activity containing (a) the functions called by it and (b) its accessed global variables. The format is both human readable and can also be parsed easily.

Identification of accesses to global variables and shared resources was done manually. Assistance could be provided by the MAP file generated by the linker; however, this file does not contain the functions accessing the global variables. CScope² can indeed provide assistance because it can list the accessing functions for a global variable, however, a list of variables accessed by a function is not available. In parallel to our code analysis, a tool for the analysis of such dependencies was being developed by Rapita Systems which will be commercialized in near future.

The result of the first phase of the parallelization approach is a set of APDs for the main loop. Taking aside periodic tasks executed by the scheduler (see Section 3), 11 instances of *Task Parallelism* remain executing in total 61 activities. These instances are partly nested up to a level of 4, i.e., in a *Task Parallelism* instance another (or even multiple different) instance of *Task Parallelism* can be executed in parallel. If all design patterns found would be executed with the highest level of parallelism possible, then 57 cores would be needed (assuming one thread is assigned to one core).

The analysis of the original source code was eased because of its structure. Functions are always written in a way that they deal only with single parts of the machine. Hence it was relatively easy to reveal this inherent parallelism.

4.2 Optimizing Parallelism

The aim of the second phase is adjusting the parallelism discovered in the first phase (Section 4.1) to the target architecture. The main advantage of doing the optimization in a model, too, is that the labor-intensive implementation task can be delayed for an already strongly optimized model of the software, hence (hopefully) reducing tuning efforts afterwards.

For a quite uniform many-core processor like our target architecture optimization of parallelism is interpreted as selecting a set of PDPs for parallel execution and assigning cores

²Homepage: <http://cscope.sourceforge.net/>

to them³. For this, a parameter is defined for each PDP found in the first phase. This parameter specifies the number of threads used for the execution of the pattern. Its value can vary between 1 and the number of activities contained in the pattern.⁴ Two objective values can be calculated in the model for each such configuration:

- The **approximated execution time** for a PDP helps estimating the impact of parallelism on the execution time. It is based on the WCET number gained with RapiTime on the original single-core platform. It is assumed that the execution of multiple program parts in parallel takes as long as the longest WCET of a single program part. On the one hand, this approximation can only be very rough because synchronization effort for multiple activities (and their shared variables) executed in parallel is very hard to estimate. On the other hand, the parallel execution overhead for each pattern can easily be assumed as fixed value because with algorithmic skeletons, which will be used for the implementation of PDPs, the overhead should tend towards a fixed value.
- The **number of cores** is the number of cores necessary for executing the configuration with the specified degree of parallelism.

Because of the large design space ($\approx 110 * 10^6$ possible configurations) a heuristic non-standard genetic algorithm is applied for a multi-objective exploration minimizing both the approximated execution time and the number of necessary cores. The tool developed especially for this task reads the APD of the control loop from an XML file.

In a first setup all parallelization overhead is ignored. Figure 1(a) shows all configurations evaluated by the algorithm; the best points in terms of Pareto-optimality are those which are the most to the left and the bottom, i.e., which have minimal core count and as short as possible approximated execution time. The Pareto front, which consists of the best possible configurations, contains configurations with 1 to 10 cores. This means that with 11 up to 57 cores no smaller approximated execution time than 697,295 cycles can be reached, which is equal to a speedup of 8.4 for 10 cores.

Parallelization overheads are considered in a second setup. In source code, a *Task Parallelism* PDP can be implemented by an instance of an algorithmic skeleton. From preliminary evaluations of our *Timing Analyzable Skeletons (TAS)*, which are focused on enabling static WCET with the OTAWA toolset [BCRS11], it is known to us that the main overheads are for initializing a skeleton instance and that execution time increases for every participating thread. Therefore, the approximation of execution times was refined by (a) a latency of 100,000 cycles for every parallel “execution” of a design pattern and (b) 10,000 cycles latency for each thread of a design pattern.

The comparison of the resulting speedups (see Figure 1(b)) shows an increasing reduction of the speedup with growing core numbers if latencies are respected. This is because for more cores more and more patterns need to be executed in parallel and hence also the

³Placement of shared memory structures onto the different cores is ignored for now.

⁴If a pattern is assigned less threads than activities are contained in it, then these activities (and also sub-patterns) are grouped by their approximated execution time with a greedy algorithm leading to more or less equal execution times for all threads.

acceleration is reduced by the additional latencies. The best possible speedup is 5.73 for 17 cores.

Efficiency, understood as speedup per core or achieved ratio of the maximum possible speedup, is at around 0.8 for up to 10 cores when latencies are ignored and with latencies quickly reaches around 0.6 for 10 cores and drops linearly to 0.3 for additional cores (see Figure 1(c)).

4.3 Towards a Parallel Implementation

The implementation effort, i.e., the transition from the optimized model to source code, is mainly defined by the need for (a) implementing the PDPs and (b) securing accesses to global shared variables, data, and devices (resources, in general) with locks. With TAS, the implementation of the *Task Parallelism* instances becomes quite simple.

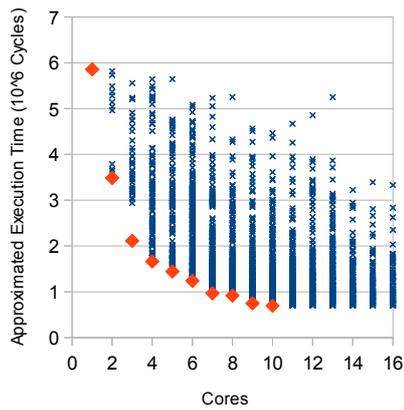
Adding locks for shared resources, in contrast, can be frustratingly complex in some cases. For each shared variable, which is accessed by two concurrently executed threads, all accesses in these threads have to be found and locks have to be placed. An evaluation showed that with reduction of the estimated execution time the number of concurrently accessed shared variables grows nearly linearly (Figure 1(d)). Adding synchronization is strongly simplified if *mutator methods* (like `get/set`) are used to access and change shared variables; in this case the necessary synchronization can often be placed in these methods.

5 Conclusion and Outlook

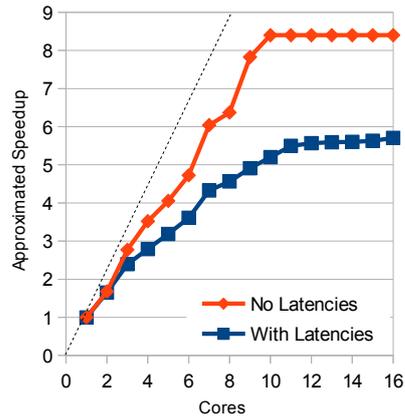
To provide additional computation power one possibility is to migrate control codes to embedded multi-core processors. For evolving existing software to a multi-threaded program necessary to exploit the newly available resources the pattern-supported parallelization process [JGU13a] was presented making strong use of parallel design patterns (PDPs).

In this paper, the model-based part is applied on a real-world code, the control code of a complex construction machine. More concise, the source code was analyzed and a UML-like model of software is built for the main control loop exhibiting 11 instances of the *Task Parallelism* PDP executing 61 activities. For these activities, accesses to more than 130 shared variables were isolated. Besides this main loop, only periodic tasks were found which are taken aside and executed by a scheduler on two dedicated cores (due to high MOETs).

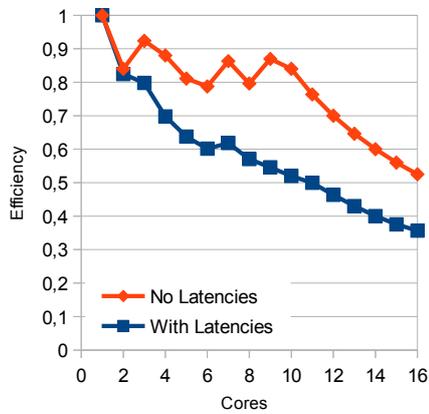
The model was refined by assigning a variable to each *Task Parallelism* instance defining the number of threads per pattern. Based on WCET numbers gathered with RapiTime on the single core TriCore processor an approximation of the overall execution time was determined also providing the number of necessary cores.



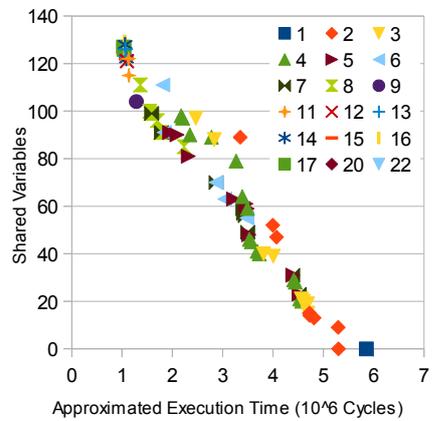
(a) Optimization for minimal approximated execution time and minimal number of cores (cut at 16). x are all evaluated configurations, \diamond are optimal configurations representing the Pareto front.



(b) Approximated optimal speedups for different numbers of cores with/without parallelization overheads



(c) Approximated efficiency, i.e., speedup per core, with/without parallelization overheads



(d) Approximated execution time with parallelization overheads and the number of shared variables (“worst-case”) necessary for the implementation with different numbers of cores

Figure 1: Evaluation results from multi-objective optimization of the model of the main control loop

For the main loop without any code parts being executed periodically, the approximated maximum speedup is 5.7 with overheads for every multi-threaded “execution” of a PDP instance and every participating thread (speedup of 8.4 without any overheads). The Pareto-optimal configurations found by this model-based optimization can be the basis for changing the original source code.

For the execution of periodic tasks two cores are necessary (see Section 3). From a processor with 16 cores, 14 cores would be available for the execution of the main loop. This is enough to achieve a near-maximum speedup.

The effort for the analysis of the original source code was high because tool support for the identification of accesses to shared variables was not yet fully available. The effort for implementation is mainly influenced by the number of global variables to which accesses must be synchronized. An approximation of the number of such variables showed that their number grows linearly with the decrease of the approximated execution time.

Preliminary results from implementing the design patterns with algorithmic skeletons show promising results. This work will be continued and the model for the approximation of execution times will be refined. Also a static WCET analysis of the code base with OTAWA will be tackled.

Acknowledgments

The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement no. 287519 (parMERASA).

References

- [ATB93] Neil Audsley, Ken Tindell, and Alan Burns. The end of the line for static cyclic scheduling. In *Proceedings of the 5th Euromicro Workshop on Real-time Systems*, pages 36–41. Society Press, 1993.
- [AUT13] AUTOSAR. Specification of Operating System V5.2.0. Technical report, AUTOSAR, 2013. http://www.autosar.org/download/R4.1/AUTOSAR_SWS_OS.pdf.
- [BCRS11] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. *OTAWA: An Open Toolbox for Adaptive WCET Analysis*, volume 6399 of *Lecture Notes in Computer Science*, pages 35–46. Springer, Berlin and Heidelberg, 2011.
- [CKW⁺13] Hyoun Kyu Cho, Terence Kelly, Yin Wang, Stéphane Lafortune, Hongwei Liao, and Scott Mahlke. Practical lock/unlock pairing for concurrent programs. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–12, Los Alamitos, CA, USA, 2013. IEEE Computer Society.
- [CM12] Daniel Cordes and Peter Marwedel. Multi-objective aware extraction of task-level parallelism using genetic algorithms. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, pages 394–399, 2012.

- [CMM10] Daniel Cordes, Peter Marwedel, and Arindam Mallik. Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming. In *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 267–276, October 2010.
- [Col04] Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, March 2004.
- [Fos95] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
- [GJU13] Mike Gerdes, Ralf Jahr, and Theo Ungerer. parMERASA Pattern Catalogue: Timing Predictable Parallel Design Patterns. Technical Report 2013-11, Department of Computer Science at the University of Augsburg, 2013.
- [GL13] Urs Gleim and Marcus Levy. MTAPI: Parallel Programming for Embedded Multicore Systems, 2013. http://www.multicore-association.org/pdf/MTAPI_Overview_2013.pdf.
- [GWG⁺11] Mike Gerdes, Julian Wolf, Irakli Guliashvili, Theo Ungerer, Michael Houston, Guillem Bernat, Stefan Schnitzler, and Hans Regler. Large drilling machine control code – Parallelisation and WCET speedup. In *6th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 91–94, June 2011.
- [JGU13a] Ralf Jahr, Mike Gerdes, and Theo Ungerer. On Efficient and Effective Model-based Parallelization of Hard Real-Time Applications. In *Schloss Dagstuhl Tagungsband des neunten Workshops über Modellbasierte Entwicklung eingebetteter Systeme*, MBEES, pages 50–59, Munich, 2013. fortiss GmbH.
- [JGU13b] Ralf Jahr, Mike Gerdes, and Theo Ungerer. A Pattern-supported Parallelization Approach. In *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '13, pages 53–62, New York, NY, USA, 2013. ACM.
- [KMMS10] Kurt Keutzer, Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. A design pattern language for engineering (parallel) software: merging the PLPP and OPL projects. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, ParaPLoP '10, pages 9:1–9:8, New York, NY, USA, 2010. ACM.
- [MSM04] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, first edition, 2004.
- [ORS13] Haluk Ozaktas, Christine Rochange, and Pascal Sainrat. Automatic WCET Analysis of Real-Time Parallel Applications. In *13th International Workshop on Worst-Case Execution Time Analysis*, volume 30, pages 11–20, Dagstuhl, Germany, 2013.
- [Sut05] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs's Journal*, 30(3):202–210, 2005.
- [Sys] Rapita Systems. RapiTime Explained. White Paper MC-WP-001-17. <http://www.rapitasystems.com/downloads/white-papers/rapitime-explained>.

Preliminary Experience of using mbeddr for Developing Embedded Software

Markus Voelter

independent/itemis
Oetztaler Strasse 38
70327 Stuttgart
voelter@acm.org

Abstract: Over the last few years, and as part of the LW-ES KMU Innovativ research project, a team of developers at itemis and fortiss have developed the mbeddr system, which relies on language engineering to build a new class of environment for embedded software development. In essence, mbeddr consists of a set of extensions to C (such as state machines, units, interfaces and components) as well as a few additional languages for requirements engineering, documentation and product line engineering. mbeddr is still new, but a number of systems have been built with mbeddr. In this paper I summarize some preliminary experience with using mbeddr's default extensions to build embedded systems based on a set of case studies. The ability for mbeddr to be extended is *not* discussed in this paper, even though this has proven very useful as well.

1 About mbeddr

mbeddr¹ is an open source project supporting embedded software development based on incremental, modular domain-specific extension of C. It also supports other languages, for example, for capturing requirements, writing documentation that is closely integrated with code, and for specifying product line variability. Figure 1 shows an overview, details are discussed in [VRKS13] and [VRSK12]. mbeddr builds on the JetBrains MPS language workbench², a tool that supports the definition, composition and integrated use of general purpose or domain-specific languages. MPS uses a projectional editor, which means that, although a syntax may look textual, it is not represented as a sequence of characters which are transformed into an abstract syntax tree (AST) by a parser. Instead, a user's editing actions lead *directly* to changes in the AST. Projection rules render a concrete syntax *from* the AST. Consequently, MPS supports non-textual notations such as tables or mathematical symbols, and it also supports wide-ranging language composition and extension [Voe11] – no parser ambiguities can ever occur when combining languages.

mbeddr comes with an extensible implementation of the C99 programming language. On top of that, mbeddr ships with a library of reusable extensions relevant to embedded soft-

¹<http://mbeddr.com>

²<http://jetbrains.com/mps>

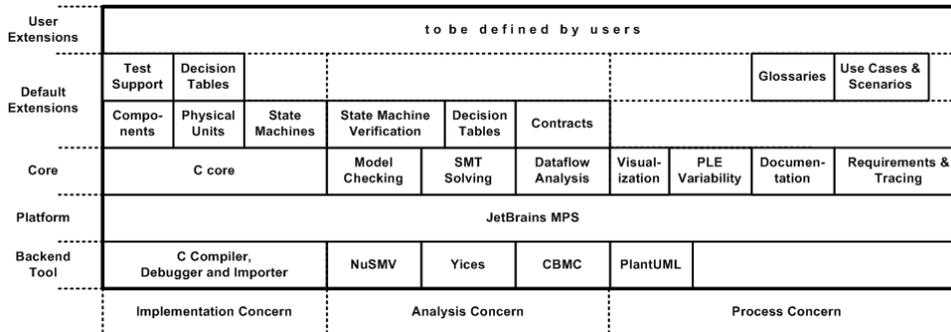


Figure 1: The mbeddr stack rests on the MPS language workbench. The first language layer contains an extensible version of C plus special support for logging/error reporting and build system integration. On top of that, mbeddr introduces default C extensions.

ware. As a user writes a program, he can import language extensions from the library and use them in his program. The main extensions include test cases, interfaces and components, state machines, decision tables and data types with physical units. For many of these extensions, mbeddr provides an integration with static verification tools [RVSK12]. mbeddr also supports several important aspects of the software engineering process: documentation, requirements and product line variability. These are implemented in a generic way to make them reusable with any mbeddr-based language (we discuss aspects of the requirements support in detail in the remainder of this paper). Finally, users can build extensions to any of the existing languages or integrate additional DSLs.

2 Challenges

This section introduces a few specific challenges in embedded software. They are the motivation for building mbeddr. This specific set of challenges is derived from industry experience of the mbeddr team; however, the challenges are in line with those reported by other authors from different communities (representative examples are [SK01, Lee00, Lee08, Bro06, KMT12]).

Abstraction without Runtime Cost Domain-specific abstractions provide more concise descriptions of the system under development. Examples in embedded software include data flow blocks, state machines, or interfaces and components. For embedded software, where runtime footprint and efficiency is a prime concern, abstraction mechanisms are needed that can be resolved before or during compilation, and not at runtime.

C considered Unsafe While C is efficient and flexible, several of C's features are often considered unsafe. For example, unconstrained casting via `void` pointers, using `ints` as Booleans, the weak typing implied by `unions` or excessive use of macros can result in runtime errors that are hard to track down. Consequently, these unsafe features of C must be prohibited in safety-critical domains.

Program Annotations For reasons such as safety or efficiency, embedded systems often require additional data to be associated with program elements. Examples include

physical units, coordinate systems, data encodings or value ranges for variables.

Static Checks and Verification Embedded systems often have to fulfil safety requirements. Standards such as ISO-26262, DO-178B or IEC-61508 require that for high safety certification levels various forms of static analyses are performed on the software. More and more embedded software systems have to fulfil strict safety requirements [LT09].

Process Support There are at least three cross-cutting and process-related concerns relevant to embedded software development. First, many certification standards (such as those mentioned above) require that code be explicitly linked to requirements such that full traceability is available. Second, many embedded systems are developed as part of product-lines with many distinct product variants, where each variant consists of a subset of the (parts of) artifacts that comprise the product-line. This variability is usually expressed using the C preprocessor via `#ifdefs`. As a consequence, variant management is a huge source of accidental complexity. The third process-related concern is documentation. In most projects, various software architecture and design documents have to be created and kept in sync with the code. If they are created using Word or \LaTeX , no actual connection exists between the documentation and the code. This is tedious and error prone. Better tool support is urgently required.

3 Example Systems and their Use of mbeddr

To validate the usefulness of mbeddr based on the challenges above, this paper relies on the experiences from a number of development projects run with mbeddr³. The projects range from demo applications to real-world development projects:

Smartmeter: The Smartmeter project is the first commercial use of mbeddr and targets the development of the software for a 3-phase smartmeter. The software comprises ca. 40,000 lines of mbeddr code, has several time-sensitive parts that require a low-overhead implementation and will have to be certified by the future operator. This leads to an emphasis on performance, testing, formal analyses and requirements tracing. The software exploits existing code supplied by the hardware vendor in the form of header files, libraries and code snippets, even though most of the system has been rewritten by now. Smartmeter is developed by itemis France.

Park-o-Matic: As part of the LW-ES project, BMW Car IT⁴ has developed an AUTOSAR component based on mbeddr. This component, called Park-o-Matic⁵, coordinates various sensors when assisting the driver to park the car. It is fundamentally a state-based system. As part of this project, AUTOSAR-specific generators had to be built for the mbeddr components language. The current mbeddr generators map the components to plain C. In case of AUTOSAR, components have to integrate with the runtime environment (RTE), which means, for example, that calls on required port have to be translated to AUTOSAR-specific macro calls. In addition, an XML file has to be generated that describes the software component so that it can be integrated

³Separate documents for some of these can be found at <http://mbeddr.com/learn.html>

⁴<http://www.bmw-carit.com/>

⁵This is not the actual name; I was not allowed to use the real name in the thesis.

with others by an integration tool.

Lego Mindstorms: This example looks at the first significant demonstration project built with mbeddr: a set of C extensions for programming Lego Mindstorms⁶ robots. These robots can be programmed with various approaches and languages, among them C. In particular, there is an implementation of the OSEK⁷ operating system called Lejos OSEK⁸. We have developed several robots (and the respective software) based on a common set of C extensions on top of Lejos OSEK. Since OSEK is also used outside of Lego Mindstorms for real-world embedded applications, this system has relevance beyond Lego. The system was developed by the mbeddr team.

Pacemaker: This system, developed by students at fortiss, addresses mbeddr's contribution to the Pacemaker Challenge⁹, an international, academic challenge on the development and verification of safety-critical software, exemplified by a pacemaker. This system emphasizes code quality, verification techniques and systematic management of requirements. Performance is also important, since the software must run on the very limited resources provided by the microcontroller in a pacemaker.

4 Addressing the Challenges

This section revisits the challenges introduced in Section 2 to show how mbeddr's features address these challenges.

4.1 Abstraction without Runtime Cost

This section investigates whether and how mbeddr's extensions are used in the example systems.

Smartmeter: Smartmeter uses mbeddr's C extensions extensively. It uses mbeddr's components to encapsulate the hardware-dependent parts of the system. By exchanging the hardware-dependent components with stubs and mocks, integration tests can be run on a PC without using the actual target device. As a side effect, the software can be debugged on a normal PC, using mbeddr's debugger. While this does not cover all potential test and debugging scenarios, a significant share of the application logic can be handled this way. In particular, interfaces and components are used heavily to modularize the system and make it testable. 54 test cases and 1,415 assertions are used. Physical units are used heavily as well, with 102 unit declarations and 155 conversion rules. The smartmeter communicates with its environment via several different protocols. So far, one of these protocols has been refactored to use a state machine. This has proven to be much more readable than the original C code. The Smartmeter team reports significant benefits in terms of code quality and robustness. The developers involved in the project had been thinking in terms of interfaces and components before; mbeddr allows them to express these notions directly in code.

⁶<http://mindstorms.lego.com/>

⁷<http://en.wikipedia.org/wiki/OSEK>

⁸<http://lejos-osek.sourceforge.net/>

⁹<http://sqlr.mcmaster.ca/pacemaker.htm>

Park-o-Matic: The core of Park-o-Matic is a big state machine which coordinates various sensors and actuators used during the parking process. The interfaces to the sensors and actuators are implemented as components, and the state machine lives in yet another component. By stubbing and mocking the sensor and actuator components, testing of the overall system was simplified.

Lego Mindstorms: mbeddr's components have been used to wrap low-level Lego APIs into higher-level units that reflect the structure of the underlying robot, and hence makes implementing the application logic that controls the robot much simpler. For example, a interface **DriveTrain** supports a high-level API for driving the robots. We use pre- and post-conditions as well as a protocol state machine to define the semantics of the interface. As a consequence of the separation between specification (interface) and implementation (component), testing of line-following algorithms was simplified. For example, the motors are encapsulated into interfaces/-components as well. This way, mock implementations can be provided to simulate the robot without using the Mindstorms hardware and API. The top-level behavior of a line-follower robot was implemented as a state machine. The state machine calls out to the components to effect the necessary changes in direction or speed.

Pacemaker: The default extensions have proven useful in the development of the pacemaker. Pacemaker uses mbeddr's components to encapsulate the hardware dependent parts. Furthermore, the pulse generator system is divided into subsystems according to the disease these subsystems cure. The pacemaker logic for treating diseases is implemented as a state machine. This makes the implementation easier to validate and verify (discussed in Section 4.4). Requirements tracing simplifies the validation activities.

Generating code from higher-level abstractions may introduce performance and resource consumption overhead. For embedded software, it is unacceptable for this overhead to be significant. It is not clearly defined what "significant" means; however, a threshold is clearly reached when a new target platform is required to run the software, "just because" better abstractions have been used to develop it, because this will increase unit cost. As part of mbeddr development, we have not performed a systematic study of the overhead incurred by the mbeddr extensions, but preliminary conclusions can be drawn from the existing systems:

Smartmeter: The Smartmeter code runs on the intended target device. This means the overall size of the system (in terms of program size and RAM use) is low enough to work on the hardware that had been planned for use with the native C version.

Pacemaker: The Pacemaker challenge requires the code to run on a quite limited target platform, the PIC18¹⁰). The C code is compiled with a proprietary C compiler. The overhead of the implementation code generated from the mbeddr abstractions is small enough so that the code can be run on this platform in terms of performance, program size and RAM use.

¹⁰http://en.wikipedia.org/wiki/PIC_microcontroller

mbeddr's extensions can be partitioned into three groups. The first group has no consequences for the generated C code at all, the extensions are related to meta data (requirements tracing) or type checks (units). During generation, the extension code is removed from the program.

The second group are extensions that are trivially generated to C, and use at most function calls as indirections. The resulting code is similar in size and performance to reasonably well-structured manually written code. State machines (generated to functions with `switch` statements), unit value conversions (which inline the conversion expression) or unit tests (which become `void` functions) are an example of this group.

The third group of extensions incurs additional overhead, even though mbeddr is designed to keep it minimal. Here are some examples. The runtime checking of contracts is performed with `if` statements that check the pre- and post-conditions, as well as assignments to and checks of variables that keep track of the protocol state. Another example is polymorphism for component interfaces, which use an indirection through a function pointer when an operation is called on a required port.

In this third group of extensions there is no way of implementing the feature in C without overhead. The user guide points this out to the users, and they have to make a conscious decision whether the overhead is worth the benefits in flexibility or maintainability. However, in some cases mbeddr provides different transformation options that make different trade-offs with regards to runtime overhead. For example, if in a given executable, an interface is only provided by one component and hence no runtime polymorphism is required, the components can be connected statically, and the indirection through function pointers is not necessary. This leads to better performance, but also limits flexibility.

We conclude that mbeddr generates reasonably efficient code, both in terms of overhead and performance. It can certainly be used for soft realtime applications on reasonably small processors. We are still unsure about hard realtime applications. Even though Smartmeter is promising, more experience is needed in this area. In addition, additional abstractions to describe worst-case execution time and to support static scheduling are required. However, these can be added to mbeddr easily (the whole point of mbeddr is its extensibility), so in the long term, we are convinced that mbeddr is a very capable platform for hard realtime applications.

Summing up, the mbeddr default extensions have proven extremely useful in the development of the various systems. Their tight integration is useful, since it avoids the mismatch between various different abstractions encountered when using different tools for each abstraction. This is confirmed by the developers of the Pacemaker, who report that *the fact that the extensions are directly integrated into C as opposed to the classical approach of using external DSLs or separate modeling tools, reduces the hurdle of using higher-level extensions and removes any potential mismatch between DSL code and C code.*

4.2 C considered Unsafe

The mbeddr C implementation already makes some changes to C that improve safety. For example, the preprocessor is not exposed to the developer; its use cases (constants, macros,

`#ifdef`-based variability, `pragmas`) have first-class alternatives in `mbeddr` that are more robust and typesafe. Size-independent integer types (such as `int` or `short`) can only be used for legacy code integration; regular code has to use the size-specific types (`int8`, `uint16`, etc.). Arithmetic operations on pointers or `enums` are only supported after a cast; and `mbeddr C` has direct support `boolean` types instead of treating integers as `Booleans`.

Smartmeter: `Smartmeter` is partially based on code received from the hardware vendor. This code has been refactored into `mbeddr` components; in the process, it has also been thoroughly cleaned up. Several problems with pointer arithmetics and integer overflow have been discovered as a consequence of `mbeddr`'s stricter type system.

More sophisticated checks, such as those necessary for MISRA-compliance can be integrated as modular language extensions. The necessary building blocks for such an extension are annotations (to mark a module as MISRA-compliant), constraints (to perform the required checks on modules marked as MISRA-compliant) as well as the existing AST, type information and data flow graph (to be able to implement these additional checks).

Finally, the existing extensions, plus those potentially created by application developers, let developers write code at an appropriate abstraction level, and the unsafe lower-level code is generated, reducing the probability of mistakes.

Smartmeter: `Smartmeter` combines components and state machines which supports decoupling message parsing from the application logic in the server component. Parsing messages according to their definition is notoriously finicky and involves direct memory access and pointer arithmetics. This must be integrated with state-based behavior to keep track of the protocol state. State machines, as well as declarative descriptions of the message structure¹¹ make this code much more robust.

4.3 Program annotations

Program annotations are data that improves the type checking or other constraints in the IDE, but has no effect on the binary. Physical units are an example of program annotations.

Smartmeter: Extensive use is made of physical units; there are 102 unit declarations in the `Smartmeter` project. `Smartmeters` deal with various currents and voltages, and distinguishing and converting between these using physical units has helped uncover several bugs. For example, one code snippet squared a temperature value and assigned it back to the original variable (`T = T * T;`). After adding the unit `K` to the temperature variable, the type checks of the units extension discovered this bug immediately; it was fixed easily. Units also help a lot with the readability of the code.

As part of `mbeddr`'s tutorials, an example extension has been built that annotates data structures with information about which layer of the system is allowed to write and read these values. By annotation program modules with layer information, the IDE can now check basic architectural constraints, such as whether a data element is allowed to be written from a given program location.

In discussions with a prospective `mbeddr` user other use cases for annotations were dis-

¹¹This is an extension that is being built as this thesis is written.

covered. Instead of physical units, types and literals could be annotated with coordinate systems. The type checker would then make sure that values that are relative to a local coordinate system and values that are relative to a global coordinate systems are not mixed up. In the second use case, program annotations would have been used to represent secure and insecure parts of a crypto system, making sure that no data ever flows from the secure part to the insecure part. Both customer projects did not materialize, though.

4.4 Static Checks and Verification

Forcing the user to use size-specific integer types, providing a **boolean** type instead of interpreting integers as Boolean, and prohibiting the preprocessor are all steps that make the program more easily analyzable by the built-in type checker. The physical units serve a similar purpose. In addition, the integrated verification tools provide an additional level of analysis. By integrating these tools directly with the language (they rely on domain-specific language extensions) and the IDE, it is much easier for users to adopt them.

Smartmeter: Decision tables are used to replace nested **if** statements and the completeness and determinism analyses have been used to uncover bugs. The protocol state machines are model-checked. This uncovered bugs introduced when refactoring the protocol implementation from the original C code to mbeddr state machines.

Pacemaker: The core behavior of the pacemaker is specified as a state machine. To verify this state machine and to prove correctness of the code, two additional C extensions have been developed. One supports the specification of nondeterministic environments for the state machine (simulating the human heart), and other one allows the specification of temporal properties (expressing the correctness conditions in the face of its nondeterministic environment). All three (state machine, environment and properties) are transparently translated to C code and verified with CBMC.

Park-o-Matic: It was attempted to use formal analyses for verifying various aspects of the state machine. However, this attempt failed because the analyses were only attempted *after* the state machine was fully developed, at which point it was tightly connected to complex data structures via complex guard conditions. This complexity thwarted the model checker.

The overall experience with the formal analyses is varied. Based on the (negative) experience with Park-o-Matic and the (positive) experience with Smartmeter and Pacemaker, we conclude that a system has to be designed for analyzability to avoid running into scalability issues. In Park-o-Matic, analysis was attempted for an almost finished system, in which the modularizations necessary to keep the complexity at bay were not made.

4.5 Process Support

mbeddr directly supports requirements and requirements tracing, product-line variability and prose documentation that is tightly integrated with code. This directly addresses the three process-related challenges identified before. All of them are directly integrated with the IDE, work with any language extension and are often themselves extensible. For example, new attributes for requirements or new kinds of paragraphs for documents can be

defined using the mens of the MPS language workbench on which mbeddr relies.

Smartmeter: Smartmeter uses requirements traces: during the upcoming certification process, these will be useful for tracking if and how the customer requirements have been implemented. Because of their orthogonal nature, the traces can be attached also to the new language concepts specifically developed for Smartmeter¹².

Pacemaker: Certification of safety-critical software systems requires requirements tracing, mbeddr's ubiquitous support makes it painless to use. Even though this is only an demo system for the Pacemaker Challenge, it is nonetheless an interesting demonstration how domain-specific abstractions, verification, requirements and requirements tracing fit together.

Lego Mindstorms: Lego being what it is, it is easy to develop hardware variants. We have used mbeddr's support for product-line variability to reflect the modular hardware in the software: sensor components have been statically exchanged based on feature models.

The requirements language has been proven very useful. In fact, it has been used as a standalone system for collecting requirements. Tracing has also proven to be useful, in particular, since it works out of the box with any language.

The documentation language has not been used much in the six example systems, since it is relatively new. However, we are currently in the process of porting the complete mbeddr user guide to the documentation language. The tight integration with code will make it very easy to keep the documentation in sync with an evolving mbeddr.

The experience with the product-line support is more varied. The definition of feature models and configuration works well (which is not surprising, since it is an established approach for modeling variability). The experience with mapping the variability onto programs using presence condition is mixed. It works well if the presence condition is used to remove parts of programs that are not needed in particular variants. However, once references to variable program parts get involved, the current, simple approach starts to break down. The same is true if the variability affects the type of program nodes. A variability-aware type system would be required. At this point it is not clear whether this is feasible algorithmically and in terms of performance. Also, without enhancements of MPS itself it is likely not possible to build such a variability-aware type system generically, i.e. without explicit awareness of the underlying language. This would be unfortunate, since extensions would have to be built in a variability-aware way specifically.

5 Conclusion

In this paper I have reported on some preliminary experience with using mbeddr for developing embedded software. The results so far are promising, even though more research is required. Two perspectives are obvious. First, the impact of the ability to build domain-specific extensions "on the fly" needs to be evaluated, since in this scenario, the effort and

¹²An important aspect of mbeddr is that project-specific extensions can be developed easily. However, this aspect of mbeddr is not considered in this paper.

complexity of building mbeddr extensions with MPS in becomes important. So far, most of the extensions (even those specific to Smartmeter) have been built by (or with support from) the mbeddr team. Second, the benefits of MPS for building flexible and extensible systems like mbeddr don't come quite for free: the MPS takes some time to get used to. We are currently running a survey among MPS and mbeddr users to find out more about this aspect. mbeddr has also been selected as the basis for a new embedded engineering tool by a major international tool vendor. While this is not a scientifically relevant result, it is certainly very encouraging to the mbeddr team.

Acknowledgements: Even though I am the author of this paper, mbeddr is a team effort with my colleagues Bernd Kolb, Dan Ratiu, Kolja Duffman and Domenik Pavletic. I also want to thank Stephan Eberle, Stefan Schmierer and Birgit Engelmann (for trying out mbeddr when it was not very mature yet) as well as the MPS team at JetBrains led by Alexander Shatalin (for tirelessly helping us with MPS issues and questions).

References

- [Bro06] Manfred Broy. Challenges in automotive software engineering. In *Proceedings of the 28th international conference on Software engineering, ICSE '06*. ACM, 2006.
- [KMT12] Adrian Kuhn, GailC. Murphy, and C.Albert Thompson. An Exploratory Study of Forces and Frictions Affecting Large-Scale Model-Driven Development. In RobertB. France, Juergen Kazmeier, Ruth Breu, and Colin Atkinson, editors, *Model Driven Engineering Languages and Systems*, volume 7590 of *LNCS*. Springer, 2012.
- [Lee00] E.A. Lee. What's ahead for embedded software? *Computer*, 33(9):18–26, 2000.
- [Lee08] E.A. Lee. Cyber Physical Systems: Design Challenges. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 363–369, 2008.
- [LT09] Peter Liggesmeyer and Mario Trapp. Trends in Embedded Software Engineering. *IEEE Softw.*, 26, May 2009.
- [RVSK12] Daniel Ratiu, Markus Voelter, Bernhard Schaetz, and Bernd Kolb. Language Engineering as Enabler for Incrementally Defined Formal Analyses. In *FORMSERA'12*, 2012.
- [SK01] Janos Sztipanovits and Gabor Karsai. Embedded Software: Challenges and Opportunities. In ThomasA. Henzinger and ChristophM. Kirsch, editors, *Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 403–415. Springer Berlin Heidelberg, 2001.
- [Voe11] Markus Voelter. Language and IDE Development, Modularization and Composition with MPS. In *GTTSE 2011*, LNCS. Springer, 2011.
- [VRKS13] Markus Voelter, Daniel Ratiu, Bernd Kolb, and Bernhard Schaetz. mbeddr: Instantiating a Language Workbench in the Embedded Software Domain. *Journal of Automated Software Engineering*, 2013.
- [VRSK12] Markus Voelter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. mbeddr: an Extensible C-based Programming Language and IDE for Embedded Systems. In *Proc. of the 3rd conf. on Systems, programming, and applications: software for humanity, SPLASH '12*, 2012.

Towards a Curriculum for Model-Based Engineering of Embedded Systems

Bernd-Holger Schlingloff

January 19, 2014

Abstract

Even though the theory and industrial practice of model-based techniques has reached a certain maturity, there seems to be no consensus of what should constitute a curriculum in this field. In this position paper, we report on several classes we gave on the bachelor and master level, as well as for industrial participants. We argue that there is a strong need for lightweight tools and platforms which allow to teach all relevant concepts, yet are easy to install and to use for beginners.

1 Introduction

An embedded system is a computational system which is a fixed part of a technical system. Model-based engineering is the craft of constructing technical systems from abstract models of the system's structure and behaviour. Within the last ten years, model-based engineering has become the preferred design methodology for embedded systems in automotive, aerospace, and other domains. There is a high industrial demand for skilled experts in this field. However, although a substantial body of knowledge in the field is available, there seems to be no consensus amongst academic teachers as to what constitutes the common core which should be taught at universities.

This paper is a first attempt to remedy this situation; it should be seen as a basis for discussion rather than a final proposal. We review the current status of the subject at different universities, and propose a curriculum which has been taught several times already. We discuss our experiences and the student's feedback, and give an outlook on further activities.

2 Model-Based Engineering of Embedded Systems

The subject arises from the intersection of two fields: Modelling in computer science, and embedded systems design. Both of these fields have been investigated for some time, and there is an extensive scientific literature on these subjects available [GHPS13, BJK⁺05, ZSM11]. For embedded systems, several universities have begun to start bachelor and masters programs [UFra, UFrb, TUE, UPe]. Most of these programs are composed from classes taken from the computer science and electrical engineering curriculum. Model-based engineering is taught differently in different engineering disciplines: Mechanical engineers have, e.g., classes on simulation with Simulink, electrical engineers learn, e.g., control-theoretic modelling of dynamic systems, and computer science students attend classes, e.g., on software modelling with UML.

Combining these two threads poses the challenge of selecting a coherent and consistent subset which nevertheless fits into the allotted time frame. Clearly, the union of all relevant topics could easily fill a complete bachelor and masters degree program. However, it is not agreed whether such a specialized program would be accepted by university administrations and students. Therefore, we constrain ourselves to the discussion of one module of two or four hours per week, with additional lab classes and maybe a specialized continuation in a subsequent semester.

Such a module could be part of any appropriate bachelor or master program in the engineering sciences. We do not want to restrain ourselves to a particular discipline such as electrical, mechanical, or software engineering. That is, we assume that the module shall be adapted to the specific prerequisite knowledge of the audience. For example, most computer science students have heard about UML in their second year, but have only vague knowledge of differential calculus. Mechanical engineers know about finite element methods, but have not heard about code generation techniques. Electrical engineers have learned about electronic circuit analysis, but may not be proficient in systematic test design. Therefore, the curriculum of the module needs to be adjusted to the department and prior skills of the participants.

Via the current Bologna process for the comparability of higher education qualifications, European degree programs are harmonized to uniform bachelor and master studies. Here, the proposed module could be allocated, e.g., as an interdisciplinary course on the bachelor level, which can be credited in several departments. Specialized extensions can be offered for different areas on the masters level.

3 A Proposed Curriculum

A module covering the subject under discussion must convey the most important principles of both embedded systems engineering and model-based design. Qualification goals are that participants are able to design and implement a reasonably complex embedded system. Thus, the module should have a strong emphasis on concrete examples. Furthermore, since the subject is still evolving, students should get an impression on current and future trends in the field. Therefore, they should not only acquire profound skills in a particular modelling language, but also learn different modelling paradigms, and meta-modelling concepts with which to link these paradigms. Similarly, they should not only experiment with the current computational and physical hardware on which present-day embedded systems are based, but also learn about future developments, e.g., in the cyber-physical domain. Students should be able to estimate economic and social impacts of the technology. Therefore, productivity of the design method and quality of the resulting products are to be considered frequently.

Subsequently, we sketch a curriculum which has been taught twice at the Humboldt Universität zu Berlin, in various Erasmus-lectures at the University of Swansea, and in part at international summer schools in Hanoi and Thessaloniki.

1. Basic definitions

This part contains the necessary foundations; it answers questions like “what is an embedded system”, “which software engineering processes exist”, “what are the basic constituents of embedded hardware”, “what is the present and future market importance of embedded systems”, “which main design challenges exist”, etc. We also show some lab prototypes of embedded systems together with their design models in order to give a glimpse ahead.

2. Requirements engineering

Here we discuss issues like the difference between user specification and techni-

cal specification, present some examples of industrial requirements documents, and discuss methods for requirements elicitation and -management. Students are also introduced to stakeholder analysis in order to understand how to capture the socio-economic aspects of a system under design.

3. **Systems modelling**

A major instrument in dealing with the design complexity is systems engineering. Students are being introduced to concepts of SysML in order to model items like the system's lifecycle, deployment, variability and product line design. Use case diagrams are used to describe the human-machine interface and interactions with other systems. A goal is to teach the students how to develop a holistic view of a system in its intended environment.

4. **Continuous modelling**

This part is a crash course in control theory. It covers the basic mathematics to describe the behavior of dynamical systems with inputs, including some linear differential equations, as well as block diagrams and continuous modelling tools such as Simulink or Scilab. Prototypical examples are an inverted pendulum as well as a two-dimensional cat-mouse race. Students learn how to clearly describe the system's boundaries, in order to distinguish between environment model and system model.

5. **Discrete modelling**

Here, we introduce the classical models of software engineering: structural and behavioral diagrams for the static and dynamic description of systems. We take care to emphasize that there is a large variety of well-established modelling languages, each in its own right. For practical demonstration purposes, we focus on UML class and component diagrams, communication and sequence diagrams, and state machine diagrams. Students learn how to come up with a first conceptual model, and how to refine this abstract model to a concrete one in various steps.

6. **Code generation**

This part describes techniques to generate Java or C code from various models. We describe "human readable" and "machine optimized" code generation transformations for both block-diagram and state-transition models. Furthermore, we discuss how to build an actual running prototype by linking abstract events to concrete IO ports. An example is a blinking LED on an experimental board which can be interrupted by a push button. The main message for students is that "the theory actually works", i.e., that model based design is not a theoretical possibility but a practical engineering method.

7. **Meta-modelling**

We introduce meta-modelling concepts as a generalization of code generation: whereas a code generator is just a model-to-text transformer, general model transformations are able to link different types of models. We use the MOF and QVT formalisms to demonstrate the concepts. We also include a discussion of domain-specific modelling languages such as ladder logic or function block diagrams for PLC programming. The idea is to enable students to extend their understanding also to modelling paradigms which are not mentioned in the course.

8. **Embedded platforms**

This part gives a short survey of embedded hardware, as well as real-time operating systems. Topics include microprocessor architectures, FPGA and ASIC programming, system-on-chip and embedded CPUs, communication methods,

power consumption, etc. On the software side, scheduling, resource allocation and process communication are discussed with example operating systems like RTLinux, FreeRTOS, or others. The choice of topics is mainly determined by the hardware available for the module. Luckily, with platforms like Arduino, Raspberry Pi, or Lego EV3, inexpensive working material for students can be procured. Depending on the available hardware, also current trends like Bluetooth Low Energy or Zigbee can be discussed here.

9. **Functional safety**

After the very practical implementation work in the previous part, the second part of the module is concerned with quality assurance. First, we introduce the students to the basic concepts of functional safety as defined in IEC 61508. We present the techniques of hazard and risk analysis, as well as FMEA and FTA. Students are challenged to determine the safety integrity level of a certain system, and to conclude which measures are to be taken during the design. The goal is to raise the awareness that functional safety considerations can influence all stages of the system's design.

10. **Fault tolerant design**

In close connection to the previous part, methods for fault tolerance are discussed. We pose the challenge to design a system of two communicating processors which emit a common synchronized pulse signal, such that each processor can be shut off and rebooted without interrupting the pulse. We discuss dual channeling and identify possible single points of failure on several examples. Students learn how to analyze system models with respect to safety requirements.

11. **Software and tool qualification**

This optional part discusses procedures for the assessment and qualification of software and software tools. We discuss the relevant standards, e.g., EN 50128, ISO 26262 and DO-330. Additionally, we discuss some coding standards like MISRA-C. Again, the goal is to raise awareness for safety-oriented design. To do so, we also exhibit some infamous compiler bugs and their potential effects.

12. **Model-based testing**

In this part we present methods for test derivation from behavioural models. We compare manually designed with automatically generated test suites and explain different model-based coverage criteria. We also show how to re-use test cases from model-in-the-loop via software-in-the-loop to hardware-in-the-loop tests. This brings us to debugging and simulation methods, which are only briefly mentioned.

13. **Static analysis and software verification**

This part deals with abstract interpretation and model checking. We show how to formulate and formally verify invariants for a behavioural model. We discuss the state-space explosion problem and demonstrate the limitations of the available technologies. Students should get a feeling for the capabilities of modern automatic and semi-automatic verification tools.

14. **Domain-specific methods**

The last part highlights some methods which are specific to certain application domains. Foremost is the area of automotive software engineering, which has evolved to a curriculum of its own [SZ13]. Current topics are Automotive SPICE, ISO 26262, and the AUTOSAR standardized architecture. Other domains include embedded railway techniques and medical systems, where we discuss implantable devices and body-area networks.

Another specialized domain is that of robotics and automation. Here we discuss items like virtual factories, mobile robotics and autonomous systems. We conclude with a summary and discussion of ethical responsibilities for certain industrial applications.

Lab classes

For the above lecture series it is essential that it is accompanied by suitable lab classes. Whereas the lectures present small, “ad-hoc” examples for each topic, the purpose of the lab classes is to deal with larger, coherent examples which covers several topics.

In our previous classes, we used several case studies as exercises.

- **Pedelec**
This example describes a modern bicycle with electric auxiliary motor. The task is to come up with requirements for the system and the control unit, to elaborate non-functional and safety requirements, to develop a specification for the pedal power amplification and battery management, build a system model, and to derive code for the display unit. We use IAR visual state and some SI-Labs evaluation boards for the actual implementation.
- **Pace maker**
This example is modelled after a published industrial case study [Bos07]. Modelling of this case study is treated in an accompanying text book [KHCD13]. The students are to design the basic functionality of a fault-tolerant pacemaker in certain operating modes. The example includes timing, fault tolerance and safety considerations, and shows an actual industrial requirements document.
- **Türsteuergerät**
This example is an automotive door control unit described in the literature [HP02]. The specification describes the operation of power windows, seat adjustment, and interior lightning. Students are to build structural and behavioural models for part of the functionality. Unfortunately, the requirements document is only available in German. Therefore, we plan to replace it by a more up-to-date case study which is currently being developed in the German SPES_XT project [SPE].

Potential continuation topics

The above curriculum is by no means a complete list of material which is relevant for the subject area. There are several additional topics which could be handled in specialized modules.

- **Physical design**
Currently, this area is not included in our curriculum. With the advent of 3D printers, however, it is becoming more and more interesting to also include the physical design into the model-based design cycle. For physical modelling, tools like Catia and others are being used; a trend in the tools industry is to integrate these with other model-based design tools. However, currently this is still an open issue; it remains to be seen which tools will be available and usable for a university teaching environment.
- **Sensor technology**
This is an advanced topic dealing with a large range of possible sensing techniques: from specialized flow sensors in pipes via highly accurate laser-scanners to CCD cameras and 3D image processing. In the above curriculum,

we use only simple switches and pushbuttons. In particular, we do not concentrate on the design of “smart” sensors with nontrivial signal preprocessing. The area, however, is highly relevant and of growing importance. An interesting future perspective is given by energy harvesting techniques, which allow sensors to compute independently from an external power supply or battery.

- **Communication**

As embedded systems are advancing to connected, “cyber-physical” systems, also the communication technology is becoming more and more important. Topics like the adaptation of the traditional ISO/OSI protocol stack to the needs of embedded systems could be discussed here, as well as high-speed optical links and wireless sensor networks. Special topics include technologies particular to embedded systems such as NFC and RFID, as well as technologies particular to specific application domain like car-to-car and car-to-roadside communication.

- **Profiles and extensions**

There are several specialized modelling languages for different purposes. Most notably, UML offers a profiling mechanism for the definition on new languages. SysML as a profile for systems modelling has been treated above. Further profiles to be considered are MARTE for real-time modelling, and UTP, the UML testing profile. In the modelling world outside of UML, other formalisms have been developed which are significant for the topic. In particular, EAST-ADL and AADL are used for architectural modelling in the automotive and avionic domain, and should be included in an advanced course.

- **Multi-core processing and deployment**

Even though present-day embedded systems are mostly built with traditional, 8-bit CPUs, the trend clearly will be to use up-to-date processors also for embedded tasks. Thus, a specialized topic is how to deal with multi-core (up to 16 CPUs) and many-core (with hundreds of CPUs) processors. Items to be discussed are on-chip synchronization and scheduling, memory access and package routing, and the deployment of models and tasks onto computing units.

- **Software tools for systems design**

Any software development method should be accompanied with relevant tools. Especially in the area we are dealing with there is an abundance of tools, both commercial and experimental. It is interesting to discuss these tools in a systematic way. In particular, tools for hardware/software co-design and design automation were not discussed in the above curriculum. Also, the large area of development management tools has been treated and could be an advanced topic.

- **Security**

Of growing importance is the field of systems security, e.g., protection against malevolent attacks. The Stuxnet worm is a prominent example which raised public awareness to this issue. An advanced course could cover technologies for authentication, cryptography, firewalls, etc. In embedded systems, security also means resistance against tampering and counterfeiting, as well as intellectual property protection. Specialized techniques such as design obfuscation have been developed which can be discussed here.

- **Commercial aspects**

Embedded systems are one of the areas with a huge potential for young, innovative start-up businesses. However, many of these new companies fail

since the founders have good technological, but little commercial knowledge. Here, the university could help by offering classes on market analysis and cost estimation, in particular with respect to the production and marketing of embedded systems. Courses on entrepreneurship and leadership could motivate students to found their own business in this area.

- **Cyber-physical systems**

A “perspectives” course close to the area of science fiction could be dealing with the elaboration of scenarios for future societies: How will the internet of things evolve? Will there be ambient assisted living aids in every home? Are intelligent humanoid robots an everyday perspective? What other smart environments could be imagined? It would be important to treat these questions from a scientific perspective, e.g., consider not only technological feasibility but also commercial viability of the scenarios.

4 Experiences and conclusion

We have taught courses roughly following the above curriculum twice at the Humboldt Universität zu Berlin, and are preparing a third round. We also gave tutorials for industrial customers based on the material. Moreover, the curriculum has been tried in various Erasmus-lectures held at the University of Swansea, where it supported the preparation of a special embedded systems program. Additionally, we have been teaching at international summer schools in Hanoi and Thessaloniki, where we gave classes covering part of the above curriculum. Here, we report on the experiences with these lectures.

A major issue is the availability and usability of tools. As argued above, the course is almost useless if not accompanied by appropriate exercises and lab classes. Therefore, the selection of tools is a major concern in the preparation. Here, we are facing a dilemma: Commercial tools usually have a complex licensing strategy; often, they are available for students only for a short evaluation period, with limited functionality, or after a complicated registration procedure. In contrast, public-domain tools are readily available; however, they often require nontrivial installation procedures, offer weak documentation and online support, and sometimes even contain bugs. In our lab classes, students complained that the process of “getting started” with a particular tool often was much more time-consuming than the actual work with the tool itself. An ideal workaround for this problem would be to prepare a “live CD”, where all necessary tools are pre-installed and there are small hands-on demos with detailed instructions how to use the tool. However, such a preparation is extremely time-consuming for the lecturer; it is not to be expected that this can be done on a wide scale. Moreover, since the subject area is under rapid evolution, such a live-CD would quickly become outdated; the immense effort would have to be done over and over again. Thus, we see a strong need for “lightweight” tools and platforms, which are easy to install and to get acquainted with. The Eclipse platform offers a plugin mechanism which allows to integrate different tools in the same development environment. This could be a perspective to mitigate the problems mentioned above. However, since also Eclipse itself is evolving, there is the problem how to migrate tools from one version to the next. Currently, we are investigating different strategies for this.

Another issue in our classes was the problem of sticking to the predetermined time schedule. In fact, in none of the mentioned courses we could “cover” the whole plan, since we had to “uncover” items which we thought that they should have been known in advance. A potential reason for this could be that the curriculum attracts students from very different backgrounds. Therefore, things which may

be obvious for some of the students may be completely new to others. A way to deal with this situation is to supply appropriate reading material for self-studies. Presently, we are referring the students mostly to original articles. Reading first-hand scientific literature is beyond the effort which average bachelor students are willing to invest for a module. Here, a textbook could be helpful, which describes the base technologies for each of the parts in more detail. Currently, we are discussing the possibility of an appropriate volume with a publisher.

The third issue which needs to be discussed is the rapid evolution of the field. The last decade has shown enormous achievements, and many of the topics of the curriculum are still “under construction”. Here, our approach is to split the curriculum into those parts which are more or less settled, and those which are hot and evolving. The above suggestion is a step in this direction: by separating basic and advanced topics, we are able to smoothly incorporate new trends into our curriculum. We hope that this is helpful to others as well.

References

- [BJK⁺05] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
- [Bos07] Boston Medical: PACEMAKER System Specification, 2007. http://sql1.mcmaster.ca/_SQLDocuments/PACEMAKER.pdf, accessed 2014-01-19.
- [GHPS13] Holger Giese, Michaela Huhn, Jan Phillips, and Bernhard Schätz, editors. *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IX, Schloss Dagstuhl, Germany, April 24-26, 2013, Tagungsband Modellbasierte Entwicklung eingebetteter Systeme*. fortiss GmbH, München, 2013.
- [HP02] Frank Houdek and Barbara Paech. Das Türsteuergerät – eine Beispielspezifikation. IESE-Report 002.02/D, Fraunhofer IESE, 2002.
- [KHCD13] F. Kordon, J. Hugues, A. Canals, and A. Dohet. *Embedded Systems: Analysis and Modeling with SysML, UML and AADL*. ISTE. Wiley, 2013.
- [SPE] BMBF: SPES_XT Software Plattform Embedded Systems “XT”. http://spes2020.informatik.tu-muenchen.de/spes_xt-home.html, accessed 2014-01-19.
- [SZ13] J. Schäuffele and T. Zurawka. *Automotive Software Engineering: Grundlagen, Prozesse, Methoden und Werkzeuge effizient einsetzen*. ATZ-MTZ Fachbuch. Vieweg+Teubner Verlag, 2013.
- [TUE] University of Eindhoven: Master’s program Embedded Systems. <http://www.tue.nl/en/education/tue-graduate-school/masters-programs/embedded-systems/>, accessed 2014-01-19.
- [UFra] Universität Freiburg: Bachelorstudiengang Embedded Systems Engineering. <http://www.eese.uni-freiburg.de/startseite.html>, accessed 2014-01-19.
- [UFrb] Universität Freiburg: Master of Science Embedded Systems Engineering. https://www.tf.uni-freiburg.de/studies/degree_programmes/master/mscese.en, accessed 2014-01-19.
- [UPe] University of Pennsylvania: Master of Science in Engineering in EMBEDDED SYSTEMS Curriculum. <http://www.cis.upenn.edu/grad/embedded-curriculum.shtml>, accessed 2014-01-19.
- [ZSM11] Justyna Zander, Ina Schieferdecker, and Pieter J. Mosterman, editors. *Model-based testing for embedded systems*. Computational analysis, synthesis, and design of dynamic systems. CRC Press, Boca Raton, 2011.

The Role of Models in Engineering of Cyber-Physical Systems – Challenges and Possibilities

Bernhard Schätz, fortiss GmbH
schaetz@fortiss.org

Abstract: In the ‘classical’ engineering process of embedded systems, models are most commonly used to support the early definition of the system (or parts thereof) or the environment. However, with the move from (complex) embedded systems to cyber-physical systems (CPS), we are experiencing a paradigm-shift. In contrast to traditional embedded systems, CPS are driven by three additional dimensions of complexity: The ‘cross’-dimension, with issues like cross-application domains, cross-technologies, cross-organizations, etc; the ‘self’-dimension, with issues like self-monitoring, self-adapting, self-optimizing, etc.; and the ‘live’-dimension, with issues like live-configuration, live-update, live-enhancement, etc.

Due to the necessary shift from the ‘classical’ development process in dedicated application domains with a clear distinction between Design/Implementation and Maintenance/Operation to the continuous development cycle merging these phases as well as roles like developer/operator/user, also the role of models in CPS requires a rethinking. In this contribution, we sketch how the model-based approach with construction, analysis and synthesis of models has to be adapted to support the engineering of cyber-physical systems, using the domain of smart energy systems for illustration.

1 Model-Based Engineering of Embedded Systems

The use of models as key technology in the engineering of embedded systems has become a best practice in several application domains. [BKKS11], for instance, shows – based on interviews of 187 engineers and developers from 14 different countries – that in the automotive domain functional modeling is adopted by 97% of the participants, with more than 95% generating code from these models. More than 40% of the participants use these models for a systematic validation and verification of the functionality under development. Overall, model-based engineering in these domains can reduce the development effort by 30%. A major driver of this reduction is front-loading of quality assurance by model-based verification and validation, helping to identify up to 60% of the design flaws before the implementation stage.

In these aforementioned domains – specifically automotive, aeronautics, and automation – the following main categories of models are generally used:

- *Functional models:* These models – often using a notation specific to the domain of application – describe the functionality of the system under development. In embedded systems, generally control-theoretic description techniques like Block Diagrams/Data Flow Diagrams and extensions thereof are used.
- *Platform models:* These models describe the platform the functionality is deployed to – in general describing both the HW elements including control units and buses as well as the accompanying SW stack like middleware or operating systems.
- *Environment models:* These models describe the behavior of (the part of) the environment the system under development is embedded into. In general, physical processes – like the rigid-body mechanics – controlled by the system are described, in rare cases these models also describe users of the system.

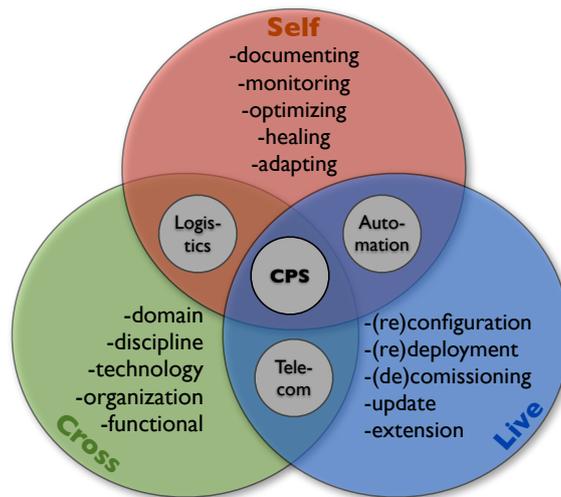


Figure 1: Dimensions of Complexity

The use of concise description techniques to explicitly **construct** these models has become one of the core assets of a model-based development process: By limiting the expressibility – and thus focusing the specification on the relevant aspects – they constructively support the assurance of quality. Furthermore, the use of explicit models enables two further mechanisms: To **analyze** models w.r.t. their validity or correctness, e.g., by simulating executable models, check for absence of nondeterminism, or verify their conformance [BLSS00]. Furthermore, to **synthesize** new models or complete partial models, e.g., by obtaining test-cases from executable models, or generate deployments of function models including execution schedules to platforms [VS13].

Due to the mentioned success of model-based development of embedded systems, and the thematic relation between embedded and cyber-physical systems, the application of this development paradigm in the later thematic field is an obvious procedure. However, as argued in Section 2, there is a substantial difference between embedded and cyber-physical systems, leading to new dimensions of complexity in the engineering process. Therefore, as sketched in Section 3, while the mechanisms of construction, analysis, and synthesis of models form the base of a CPS engineering process, additional aspects must be addressed to make them applicable.

2 Principle Characteristics of Cyber-Physical System

Before sketching how models can be used in the engineering of cyber-physical systems, it is necessary to understand the CPS complexity drivers. Clearly, we think that definitions of CPS like “systems in which computing, broadly construed, interacts with the physical world” as provided in [CPS08] do not focus on the core issues, as they also include “classical” embedded systems. In contrast, [GB12] stresses that CPS encompasses “embedded systems (...), but also logistics-, coordination- and management-processes as well as internet services”, and are “interconnected (...) locally as well as globally”. These aspects – specifically the combination of physical and organizational processes on a local and a global scale – are essential to differentiate embedded systems like a motor management system and even a complete vehicle from those CPS we consider to be a new class of system. Examples for these later class – as also listed in [GB12] – are

- *smart traffic systems* encompassing the object detection component in the individual vehicle up to the traffic management component of a large-scale telematic system

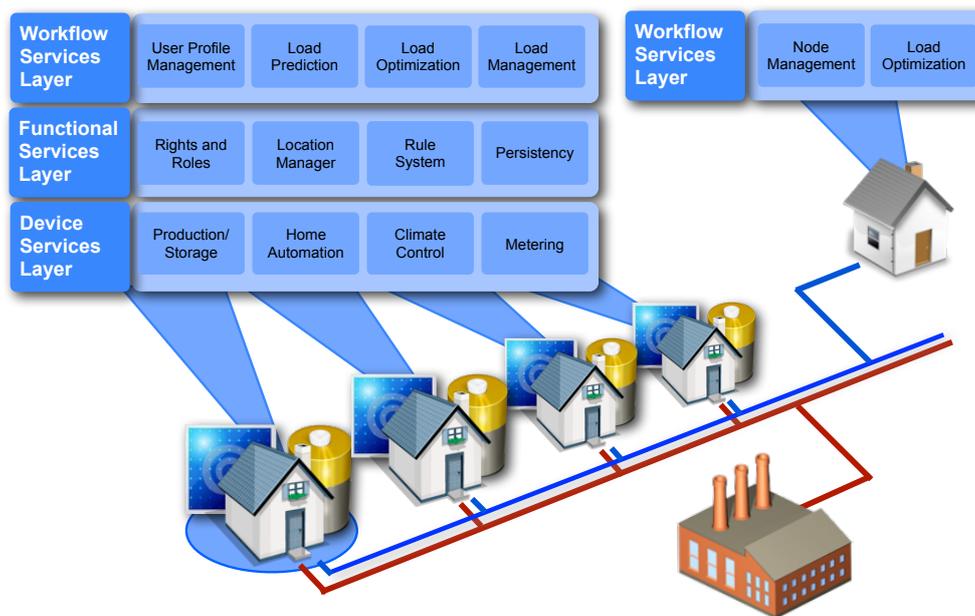


Figure 2: Smart Grid Architecture

- *smart health systems* encompassing the movement monitoring component worn by the individual patient up to the workflow component of a complete clinical information system
- *smart energy systems* encompassing the monitoring and control of a single device up to the trading of production and consumption volumes of complete regions at the spot market.

Taking a closer look at these examples makes clear that their complexity is not directly caused by covering physical and organizational as well as local and global processes, but rather by the entailed principle characteristics needed to support them. As shown in Figure 1, these characteristics can be classified into three groups, each group forming its own dimension of complexity:

- **‘Cross-Dimension:** As CPS cover large-scale processes – both physical and organizational – these processes generally go across borders, with respect to application domains, engineering disciplines, used technologies, or involved organizations, to mention a few.
- **‘Live’-Dimension:** Furthermore, CPS generally support missing critical processes, making it impossible to turn off the system to make changes and therefore, for instance, requiring (re-)configuration, (re-)deployment, (de-)commissioning, update, or enhancement during runtime.
- **‘Self’-Dimension:** Finally, being large-scale and mission critical, CPS must cooperative with system engineers, operators, users, and other systems by actively supporting their processes, requiring autonomous capabilities of documentation, monitoring, optimization, healing, or adapting, among others

While these dimensions can already be found in existing medium-scale systems – like being cross-domain and self-optimizing in logistics systems or being self-documenting and live-reconfiguring in automation systems – their massive joint presence is typical for CPS.

In the following, we use the example of a *smart energy system*, as illustrated in 2 and implemented in the EIT ICT Experience Labs Network [EIT], to illustrate these principles and also their effects on the use of models in the engineering

of CPS. As shown in the figure, a smart grid includes nodes producing, consuming, and storing energy supports, as well as enterprises aggregating loads for spot market trade, and utilities with power stations processes.

It has a substantial complexity in the ‘cross’-dimension: organizational, as each node is an individual household or enterprise; technical, as it provides services to deal with devices like batteries or PVs as well as predicting consumption or production of energy; discipline-wise, as it addresses electricity issues like net frequency as well as economy issues like price bidding; and application domain-wise like home-automation as well as grid management. Similarly, in the ‘live’-dimension: Live re-configuration and re-deployment, for instance, is support by adding devices like controllable switches or sockets; live update by changing rules like switches controlling sockets; live-enhancement by And finally in the ‘self’-dimension: Self-documentation and -monitoring allows to query the current devices of a node including their parameters; self-optimization to change the buffering schedule; self-healing to autonomously turn off and on devices according to locally available energy in island mode after circuit loss.

3 Models for the Engineering of Cyber-Physical Systems

The ‘cross’-, ‘live’-, and ‘self’-dimensions naturally influence what kind of models can be constructed, and what kind of analysis and synthesis mechanisms can be used in the engineering of CPS. In a nutshell, the most important consequences are that

- a large range models of different domains, disciplines, and technologies are used, and
- the use of models is shifted from the design and implementation phase to the operation and maintenance phase

The first consequence is caused by the ‘cross’-characteristics of CPS. As a result it is necessary to find a *common theory of modeling*, allowing to relate these different kinds of models. Often, they can be *organized in layers of models*, corresponding to the layers of services or functionalities offered by the CPS architecture. In general, each layer *includes aspects of functionality, platform, and environment* as introduced in Section 1. Since CPS are embedded in complex and large-scale physical and organizational processes, especially the later play an important role and specifically include dedicated models of the different users of a CPS.

In case of the architecture of the smart energy system shown in Figure 2, a three-tier approach of models is used. Models on the lowest layer address electrical devices of the system platform including the immediate physical processes of the environment they are embedded into. Thus, besides describing the devices of a node and their parameters, these models also include control theoretic aspects, like the relation between the monitored/controlled and input/output variables of [PM95], used to provide functionalities like sensor data fusion to mask measuring errors. Models of the middle layer add more platform information – like the location of devices – and medium-scale environment information – like effects of blinds and lights on brightness as well as user behavior – and specifically address complex functionalities like rules governing the reactions of the system to user actions. The models of the top layer allow to transcend the borders of individual nodes. On the platform and environment side, they aggregate nodes to networks of nodes including the corresponding inter-node processes like energy flow. On the functional side, models addressing aspects of optimization like (shiftable) loads or markets are added. Since the layers of the models must support the capability of the CPS to address the ‘self’-characteristics, each layer also addresses an increasing level of autonomy. Therefore, the models – most specifically of the system (functionality) and the environment – capture aspects ranging from *control* on the lower layer, to *act* on the middle layer to *plan* and *cooperate* on the top layer, as commonly found in robotic architectures [Tip05].

The second consequence – the shift of the use of models from the design/implementation phase to the operation/maintenance phase – is primarily caused by the ‘live’- and ‘self’-dimension. Since a CPS is reconfigured, updated, and enhanced during runtime, those models used in Section 1 at design time must be made available at runtime. Furthermore, since

CPS can autonomously reconfigure or adapt their behavior, especially models of their platform and functionality must be made explicit at runtime. As a result of this shift, not only the construction of these models, but also corresponding analysis and synthesis methods must be made available during operation and maintenance, turning a CPS into its own engineering and development environment.

This is of even more importance as the clear distinction between a designer/implementer and an operator/user of a system is increasingly blurred in a CPS. Using again the example of the smart energy systems, office space users are allowed to adapt the rules controlling the behavior of their office – turning them from a user into a designer. Similarly, adding a new device at runtime turns a system implementer into a system operator. As the updating or enhancement of functionality can have problematic side effects and is carried out by domain experts or users rather than system engineers, validation and verification mechanisms prior to the activation of the changed behavior are necessary. In case of the smart energy system, the rule component provides a domain-specific language to define behavior using concepts like events, values, and actions, as well as mechanism to verify the absence of flaws like overlapping or missing rules [RVMS12]. Thus, capabilities like the soundness analysis of the model-based development process mentioned in Section 1 or compatibility analysis as in [BRS13] can be integrated in a similar fashion into a CPS engineering approach. Likewise, synthesis techniques like the generation of schedules mentioned in Section 1 can be applied to support the configuration space exploration in the engineering of CPS. In the smart energy system, for instance, degradation plans for the incremental deactivation of devices in case of grid failure are synthesized from the platform and function models. Note that while these analysis and synthesis techniques require the use of *models at runtime*, these functionalities need not be provided *online* by the executing components of the CPS. For instance, in the smart energy system, the architecture uses dedicated components for the analysis and synthesis of models, which then are forwarded to the executing rule system. As however the models used are provided by the running system, both parts form a combined system, merging operating and engineering environment.

4 Summary and Conclusion

In this contribution we argued that cyber-physical systems are not only large-scale networked embedded systems, but constitute a paradigm shift in systems engineering. The key principles of cyber-physical systems and main complexity drivers are the ‘cross’-, ‘live’-, and ‘self’-domain.

Due to these principles we expect that

- the explicit use of models will play a dominant role in the engineering of CPS
- the traditional distinction between design/implementation and operation/maintenance of a system will be abandoned in favor of an integrated life cycle

However, to meet these changes in engineering CPS, we are faced with the challenges

- to provide a framework including both a common theory of modeling paradigms as well as methods to arrange (distributed) layers of models
- to turn a CPS into its own engineering and development environment, with built-in mechanisms to construct, analyze, and synthesize models of its environment, platform, and functionality

We base these claims mainly on experiences and observations on the engineering of automotive systems as well as smart energy systems, but experiences in other application domains for CPS like cooperating vehicles point to similar conclusions.

While the importance of adequate models for the engineering of CPS has also been stressed, for instance, in [GB12], in general the focus is put on the common theory of modeling paradigms. In this contribution, in contrast, we argue that besides this more foundational aspect the question on how to methodically apply models in the construction should receive more attention as their explicit use will form the basis of CPS engineering.

Acknowledgements

Many of the drawn conclusions in this contribution are based on experiences made during in the implementation of the smart energy experience lab. The authors therefore want to thank Thomas Böhm, Denis Bytschkow, Markus Duchon, Pragma Gupta, Dagmar Koss, and Zaur Molotnikov for their efforts in the implementation of the lab, as well as the EIT ICT Labs for their co-funding.

References

- [BKKS11] Manfred Broy, Sascha Kirstan, Helmut Krcmar, and Bernhard Schätz. What is the Benefit of a Model-Based Design of Embedded Software Systems in the Car Industry? In *Emerging Technologies for the Evolution and Maintenance of Software Models*. IGI Global, 2011.
- [BLSS00] P. Braun, H. Lötzbeyer, B. Schätz, and O. Slotosch. Consistent Integration of Formal Methods. In *Proc. 6th Intl. Conf on Tools for the Analysis of Correct Systems (TACAS), LNCS 2280*, 2000.
- [BRS13] Jan Olaf Blech, Harald Rueš, and Bernhard Schätz. On Behavioral Types for OSGi: From Theory to Implementation. Technical report, Computing Research Repository, 2013.
- [CPS08] Report: Cyber-Physical Systems Summit, 2008. http://iccps.acm.org/2011/_doc/CPS_Summit_Report.pdf.
- [EIT] EIT ICT Labs. Future Scenarios and Smart Energy Systems Exp. Lab Joint Web Presence. <http://www.eitictlabs.eu/fssesel>.
- [GB12] Eva Geisberger and Manfred Broy. *agendaCPS - Integrierte Forschungsagenda Cyber-Physical Systems*. Springer Vieweg/achatech, 2012.
- [PM95] D. Parnas and J. Madey. Functional Documents for Computer Systems. *Science of Computer Programming*, 1(25):41–61, October 1995.
- [RVMS12] Daniel Ratiu, Markus Voelter, Zaur Molotnikov, and Bernhard Schätz. Implementing modular domain specific languages and analyses. In *Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation*. ACM, 2012.
- [Tip05] Nils Ole Tippenhauer. A Multi-Robot Architecture for Autonomous Cooperative Behaviours. Technical report, University of Waterloo, 2005.
- [VS13] Sebastian Voss and Bernhard Schätz. Deployment and Scheduling Synthesis for Mixed-Criticality Shared Memory Systems. In *Engineering of Computer Based Systems*. IEEE, 2013.

Modellbasierte Konsistenzprüfung von Produktlinien in Automatisierungssystemen¹

Matthias Riebisch, Yibo Wang

Universität Hamburg, FB Informatik
{riebisch|wang}@informatik.uni-hamburg.de

Abstract: Bei der Entwicklung von Automatisierungssystemen stellen Produktlinien eine wichtige Technologie dar. Dazu werden Modellierungssprachen benötigt, die sowohl Variabilität und domänenspezifische Aspekte darstellen als auch Abhängigkeiten zwischen den verschiedenen Elementen von Anlage und Automatisierungssystem ausdrücken können. Vorhandene Modellierungssprachen erfüllen nur einzelne dieser Anforderungen - ein Vorgehen zur Entwicklung und Zusammenführung in einem Modell fehlt noch. Der Kurzbeitrag stellt die Entwicklung eines modellbasierten und an Produktlinien orientierten Entwicklungsvorgehens für Automatisierungssysteme im Überblick vor. Seine Evaluierung wird im industriellen Einsatz im Rahmen eines Verbundprojekts erfolgen.

1 Einleitung

Bei der Entwicklung von komplexen Systemen haben Wünsche nach kurzfristigen kundenspezifischen Anpassungen eine hohe Priorität. Bei Automatisierungssystemen besteht eine wichtige Forderung in hoher Flexibilität bezüglich der Optimierung von gesteuerten Prozessen der Verfahrens- und Fertigungstechnik. Techniken von Produktlinien beziehungsweise Systemfamilien erfüllen diese Forderungen. Diese Techniken ermöglichen außerdem die Verringerung der Entwicklungsdauer des einzelnen Automatisierungssystems, da die benötigte Variabilität bereits bei der Entwicklung der gemeinsamen Basis der Produktlinien vorgesehen wird. Variabilität wird dazu benutzt, um Entscheidungen auf eine spätere Entwicklungsphase zu verschieben werden.

Aufgrund vieler Risiken bei der Automatisierung wie Produktionsstörungen oder Umweltschäden sind frühzeitige Prüfungen erforderlich, um die Risiken zu verringern. Eine Beschreibung von Zusammenhängen und Wechselwirkungen mittels Modellen unterstützt diese Ziele. Die Komplexität, die aus den Anforderungen an zu automatisierende Prozesse und aus heterogenen Systemarchitekturen resultiert, wird durch Variabilität weiter gesteigert und erfordert daher ein modellbasiertes Vorgehen bei der Entwicklung. Jedoch sind vorhandene Vorgehensweisen und Modellierungssprachen noch nicht ausreichend entwickelt, um diese Anforderungen zu erfüllen.

Dieser Kurzbeitrag bietet einen ersten Überblick über ein Forschungsvorhaben, bei dem ein modellbasiertes und an Produktlinien orientiertes Entwicklungsvorgehen für Auto-

¹ Diese Arbeiten werden teilweise gefördert durch das BMBF unter dem Kennzeichen 01M3204C im Rahmen des Verbundprojekts „Entwurfsmethoden für Automatisierungssysteme mit Modellintegration und automatischer Variantenbewertung (EfA)“

matisierungssysteme entwickelt wird. Der wissenschaftliche Beitrag der Arbeiten besteht in der Entwicklung von domänenspezifischen Modellierungssprachen und deren Zusammenwirken sowie einem Vorgehen bei Entwicklung, Konfiguration und Verifikation von Automatisierungssystemen. Das Forschungsvorhaben ist Teil des Verbundprojekts „Entwurfsmethoden für Automatisierungssysteme mit Modellintegration und automatischer Variantenbewertung“ (EfA).

2 Stand der Technik

Für das Forschungsvorhaben sind vor allem folgende Gebiete und Vorarbeiten relevant: *Produktlinien-Ansätze* im Software Engineering haben das Featuremodell eingeführt. Features sind für den Nutzer relevante Eigenschaften eines Systems. Das Featuremodell bietet eine abstrakte Repräsentation der variablen Features sowie der Abhängigkeiten zwischen ihnen [CL01]. Auf Basis des Featuremodells ist eine kompakte Repräsentation aller Produkte einer Software-Produktlinie auf einem hohen Abstraktionsniveau für Anforderung und Grobplanung möglich [RU12]. Variabilität auf einer detaillierteren Ebene kann zum Beispiel im Familienmodell [PU04] oder durch sogenannte Assets im DOPLER-Decision-Modell [DH11] dargestellt werden. Weitere Feature-Eigenschaften und die Beziehungen zwischen Features und Artefakten können nicht dargestellt werden, so dass für Konsistenzbewertung weitere Modelle erforderlich sind, wie das Orthogonal-Variability-Modell [PO05]. Codegenerierung wird in der Automatisierungstechnik meist nicht benötigt

Modellierungssprachen mit Beschreibung von Variabilität wurden zunächst für die Software- und Systementwicklung entwickelt. Für eingebettete Systeme existieren Erweiterungen bezüglich Variabilität für Modellierungssprachen wie Simulink [PO09] und SysML [HO12]. Beide Ansätze stellen Abhängigkeiten und deren Auswirkung auf die Konsistenz von Produkten einer Produktlinie nur auf der Ebene von Features dar, nicht jedoch auf feingranularer Ebene für Bestandteile von Systemen. Eine durchgängige Konsistenzprüfung über verschiedene Modelle und Abstraktionsebenen hinweg, wie für Feature-Modell, Funktionsnetz, Funktionsbaustein und generierten Quelltext, wurde in derartigen Ansätzen nicht komplett behandelt. Unser Forschungsansatz soll dies erlauben, und so eine modellbasierte Entwicklung von Automatisierungssystemen zu erreichen.

Vorgehensweisen für Entscheidungsfindung und Konfiguration wurden ebenfalls zunächst für Software-Produktlinien entwickelt, beispielsweise in [VI10] für durchgängige Konsistenzprüfung von Feature-Modell über Komponenten-Modell bis zum Code. Für Automatisierungssysteme fehlen solche Vorgehensweisen noch.

Für die Konfiguration einer Produktlinie wird in [CH11] zwischen regelbasierten, modellbasierten und Fall-basierten Konsistenzprüfungen unterschieden. In unserem Vorhaben wird der regelbasierte Ansatz verfolgt, denn nur dieser ermöglicht es, Konsistenzregeln auf allen Ebenen im gleichen Format wie zum Beispiel mit RuleML² oder mit

² RuleML – Rule Markup Language auf Basis XML http://wiki.ruleml.org/index.php/RuleML_Home

EMFTrace³ zu definieren. Zudem sind formalisierte Regeln nach diesen Ansätzen sowohl für Menschen als auch maschinell lesbar.

3 Forschungsansatz

Für die *Modellierung von Variabilität* werden Featuremodelle angewendet. Abbildung 1 zeigt ein Beispiel aus dem Projekt EfA. Featuremodelle dienen vor allem dazu, Features und die Beziehungen zwischen ihnen (alternative, or, requires, excludes) sowie ihre Variabilität (mandatory oder optional) darzustellen. Sie werden für die Definition und die Konsistenzprüfung von Produkten auf Basis der Produktlinie angewendet. Die Modellierung von Variabilitätspunkten erfolgt nicht nur innerhalb der Featuremodelle, sondern mittels *domänenspezifischer Modellierungssprachen* auch innerhalb der Modelle für Anlage sowie Automatisierungssystem jeweils für Struktur und Verhalten.

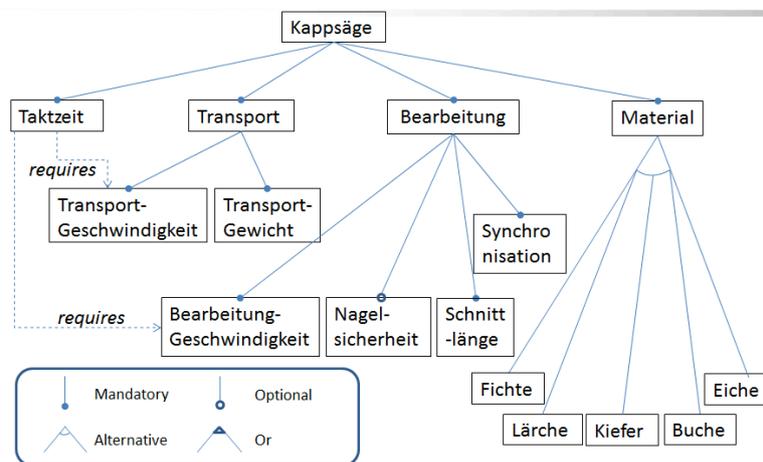


Abbildung 1: Featuremodell der Kappsäge im Projekt EfA

Entsprechende Modellierungssprachen sind in der Praxis etabliert, allerdings müssen sie um Ausdrucksmittel für Variabilitätspunkte erweitert werden. Außerdem müssen *Verknüpfungen* zwischen den verschiedenen Modellen und mit dem Featuremodell hergestellt werden, um Abhängigkeiten ausdrücken zu können, die für die Konsistenzprüfung ausgewertet werden. Zur *Zusammenführung von Modellen* bei der Entwicklung von Automatisierungslösungen wird AutomationML [AU13] als neutrales, XML-basiertes Datenformat genutzt, um die Speicherung und den Austausch zwischen verschiedenen Beteiligten und Werkzeugen bei der Planung von Automatisierungsanlagen zu ermöglichen. Abbildung 2 zeigt schematisch die mittels AutomationML beschreibbaren Aspekte. Im Projekt EfA wird Variabilität für die Aspekte Topologie und Logik betrachtet und in AutomationML modelliert. Dazu soll eine Kopplung zwischen Modellelementen von AutomationML und Features des Featuremodells vorgenommen werden. Diese Kopplung soll, wie auch bei anderen beteiligten Modellierungssprachen, über Verweise implementiert werden, die zwischen den XML-Repräsentationen der Modellierungsspra-

³ EMFTrace – Repository zum Zusammenführen von Modellen <http://sourceforge.net/projects/emftrace/>

chen etabliert werden. AutomationML wurde um neue Elemente Variation Point und Variant sowie um Attribute für Variabilität erweitert.

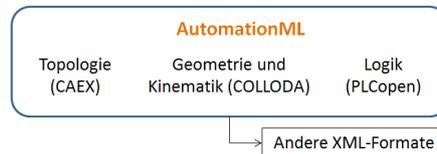


Abbildung 2: Von AutomationML abgedeckte Aspekte der Planung und Entwicklung von Automatisierungssystemen

Die Entscheidung für Art der Produktlinien-Implementierung wie 150%-Ansatz oder Delta-Ansatz [SC10] hängt vor allem von Lösungsbausteinen und Technologie der Implementierung ab. Unser Konzept sieht die Unterstützung beider Ansätze vor.

Da AutomationML keine Unterstützung für Variantenmodellierung bietet, muss es um Variationspunkte, Attribute für Varianten und Bezüge zu Features in Featuremodellen erweitert werden. Für den im Projekt angestrebten Entwurf von Automatisierungssystemen auf Basis der Produktlinie wird ein Konfigurationswerkzeug entwickelt, das auf Basis des Featuremodells die passenden Bibliothekskomponenten und zulässige Optionen für Konfigurationen anbietet. Darüber hinaus wird bei der Varianten-Konfiguration, auf Basis der vordefinierten Regeln, die Konsistenzprüfung auf jeder Detaillierungsstufe durchgeführt, um die ungültige Variantenkombination frühzeitig ausschließen zu können.

Der beim Entwurf und der Planung gewünschte Grad der Werkzeugunterstützung bestimmt den Grad der *Formalisierung der Modelle*. In semiformalen Beschreibungen wie Blockdiagrammen von SysML oder Objekt-Bäumen von AutomationML (siehe Abbildung 3) wird ein wesentlicher Teil der Semantik üblicherweise durch Bezeichner und erläuternde Texte ausgedrückt, die in eine Struktur eingebettet werden, welche durch eine formal definierte Syntax definiert ist. Ein Beispiel hierfür ist die Verbindung der Signalschnittstelle „Channel01“ zu der Schnittstelle „Start“ in Abbildung 3. Eine Prüfung, ob die Verbindung dieser Schnittstellen korrekt ist, wäre erst durch Einbeziehen von weiteren Beschreibungen zur Semantik der Blöcke möglich.

Werkzeugunterstützung zur Konsistenzprüfung setzt jedoch formalisierte Ausdrucksmittel voraus. Sind beispielsweise Konsistenzprüfungen bezüglich des Verhaltens erforderlich, werden entsprechende formalisierte Modelle für Verhalten und Logik, wie etwa erweiterte State-Charts, Petri-Netze oder Entscheidungstabellen, angewendet. Soweit der Formalisierungsgrad erhöht werden soll, werden natürlich-sprachliche Ausdrücke durch Referenzen und definierte Typen ersetzt – in unserem Forschungsvorhaben für Anlagenstruktur und logische Abhängigkeiten. Aussagen zur Semantik innerhalb anderer Modelle, wie beispielsweise zur Bedeutung eines, sollen im Rahmen des Forschungsvorhabens nicht formalisiert werden.

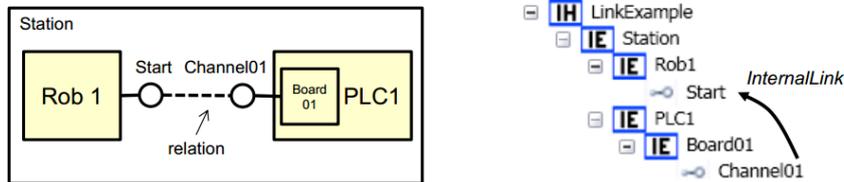


Abbildung 3: Beispiel für Blockdiagramm in SysML (links) und Objekt-Bäume in AutomationML (rechts) [AU13]

Bei Entwurfsentscheidungen im Planungsprozess für Automatisierungslösungen werden die Modelle ausgewertet, um werkzeuggestützte *Konsistenzprüfungen* durchzuführen. Im Rahmen eines Produktlinien-Ansatzes erfolgt der Entwurf überwiegend durch die Konfiguration von variablen Elementen einer Produktlinie. Konsistenzprüfungen beziehen sich dabei beispielsweise auf Beschreibungen von Attributen und Schnittstellen. Für solche Prüfungen werden Regeln entwickelt und formalisiert, die sich zum Beispiel auf Typen und Parameter von Funktionsblöcken und auf Wertebereiche von Eingangs- und Ausgangssignalen beziehen.

Als Ergebnis des Forschungsvorhabens werden die *Modelle und Regeln* eingesetzt, um Anwenderunterstützung beim Entwurf von Automatisierungslösungen auf Basis einer Produktlinie bereitzustellen. Entwickler werden beispielsweise bei Entscheidungen unterstützt, indem nur gültige Lösungsvarianten zur Auswahl angeboten und Plausibilitäts- und Konsistenzprüfungen der Entwurfsergebnisse automatisch durchgeführt werden. Der Nutzen besteht in Effizienzsteigerung sowie in Fehlervermeidung beim Entwurf von Automatisierungslösungen. Abbildung 4 stellt eine Regel für das Kappsäge-Beispiel dar. Sie prüft, ob der Markensensor in Bezug auf das Bearbeitungsaggregat korrekt platziert wurde. Durch Verweis auf einen Regel-Katalog wird gekennzeichnet, dass es sich um eine Regel für die Dimension Solution Domain und um Konsistenzprüfung bezüglich Topologie handelt. Eine Zuordnung zu Regler/Regelstrecke gibt es hier nicht.

```
<Rule RuleID="Rule01" Description="Der Markensensor muss vor der
Startposition des Bearbeitungsaggregats platziert werden">
  <Elements Type="Sensor" Alias="e1"/>
  <Elements Type="Werkzeug" Alias="e2"/>
  <Actions ActionType="ReportConsistencyViolation"
ResultType="" SourceElement="e1" TargetElement="e2"
ImpactedElement=""/>
  <Conditions>
    <BaseConditions Type="ValueGreaterThan"
Source="e1::position"
Target="e2::position"/>
  </Conditions>
</Rule>
```

Abbildung 4 Beispiel einer Konsistenzregel in EMF-Trace

Die Bewertung und Entscheidungsunterstützung wird in das *Vorgehen und die Methodik* des Entwurfs der Automatisierungssysteme integriert. Domänenspezifische sowie betriebliche Vorgaben fließen ebenfalls ein. Das Vorgehen und die Methodik definieren den Umfang und Anwendungsbereich der zu erstellenden Modelle, deren Abstimmung auf die beteiligten Rollen der verschiedenen Disziplinen, die Entwicklungsphasen von Produktlinien- und Produktentwicklung.

4 Weitere Schritte

Das hier vorgestellte Forschungsvorhaben wird seit Herbst 2012 durchgeführt und hat zu ersten Ergebnissen geführt; sein Abschluss ist für den August 2015 geplant. Als nächste Schritte werden Abhängigkeiten zwischen Elementen der benötigten Modellierungssprachen ermittelt und als Traceability-Beziehungen zwischen verschiedenen Modellen beschrieben. Weiterhin wird eine Methode zur Variantenbewertung entwickelt, die ähnlich wie die beschriebene Konsistenzprüfung Informationen aus den formalisierten Modellen ableitet. Als Implementierung dieser Konzepte wird ein prototypisches Werkzeug entwickelt. Innerhalb des Verbundprojekts EfA wird dieses Werkzeug einen Bestandteil der Werkzeugkette bilden, die beim Anforderungsmanagement auf der Sprache ReqIF⁴ basiert. Für die Anlagen-Modellierung baut sie auf AutomationML [AU13] und den daran angebotenen Planungswerkzeugen auf, für die Traceability-Links auf das Repository EMFTrace. Für das Variantenmanagement wird das Werkzeug pure::variants⁵ eingesetzt.

Literaturverzeichnis

- [AU13] -: AutomationML Whitepaper Part 1 - Architecture and general requirements. AutomationML e.V., 2013. auf <https://www.automationml.org/> am 15.01.2014
- [CL01] Clements, P.; Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley Professional, 2001; S. 114
- [CH11] Brecher, C., Karlberger, A., & Herfs, W.: Synchronisation of Distributed Configuration Tools using Feature Models. In 4th International Conference on Changeable, Agile, Reconfigurable und Virtual Production, 2011, S. 340-345
- [DH11] Dhungana, D., Grünbacher, P. & Rabiser, R.: The DOPLER meta-tool for decision-oriented variability modeling: a multiple case study. Automated Software Engineering 18, 2011; S. 77-114
- [HO12] Hoyos, H.; Casallas, R.; Jimenez, F.: Model-Based Framework for Embedded System Product Line. In Proc. 38th Annual Conference on IEEE Industrial Electronics Society (IECON), 2012; S. 3101-3106
- [PO05] Pohl, K.; Böckle, G.; Van der Linden, F.: Software Product Line Engineering – Foundations, Principles, and Techniques. Springer. 2005; S. 85
- [PO09] Polzer, A.; Kowalewski, S.; Botterweck, G.: Applying Software Product Line Techniques in Model-based Embedded Systems Engineering. In Proc. Model-based Methodologies for Pervasive and Embedded Software (MOMPES), 2009; S. 2-10
- [PU04] -: Technical White Paper Variantenmanagement mit pure::variants. pure-systems GmbH, 2004. auf <http://www.pure-systems.com/fileadmin/downloads/pv-whitepaper-de-04.pdf> am 18.02.2014
- [RU12] Russell, J.; Cohn, R.: Feature Model, Book on Demand, 2012
- [SC10] Schaefer, I.; Bettini, L.; Damiani, F.; Tanzarella, N.: Delta-oriented programming of software product lines. In Proceedings of the 14th international conference on Software product lines: going beyond (SPLC'10), Jan Bosch and Jaejoon Lee (Eds.). Springer-Verlag, Berlin, Heidelberg 2010; S. 77-91
- [VI10] Vierhauser, M.; Grünbacher, P.; Egyed, A.; Rabiser, R.; Heider, W.: Flexible and scalable consistency checking on product line variability models. In Proc. of the IEEE/ACM international conference on Automated software engineering (ASE), 2010; S. 63-72

⁴ Requirements Interchange Format (ReqIF) <http://www.omg.org/spec/ReqIF/>

⁵ pure::variants - Werkzeug zum Variantenmanagement http://www.pure-systems.com/pure_variants.49.0.html

Regelbasiertes Engineering mithilfe deklarativer Graphabfragen

Sten Grüner, Ulrich Epple

Lehrstuhl für Prozessleittechnik
RWTH Aachen University
Turmstraße 46
52064 Aachen
s.gruener@plt.rwth-aachen.de
epple@plt.rwth-aachen.de

Abstract: Die Automatisierung manueller Tätigkeiten im Bereich des Anlagenengineering ermöglicht eine effizientere Inbetriebnahme und Rekonfiguration der fertigungs- und verfahrenstechnischen Anlagen. Die Nutzung regelbasierter Systeme, deren Regeln auf deklarativen Graphabfragen aufbauen, ist eine neue Möglichkeit dieses Ziel zu erreichen. In diesem Beitrag wird ein solches System vorgestellt und für eine alltägliche Engineering-Aufgabe angewendet. Die Graphabfragen sind sowohl für Domainexperten als auch für Spezialisten aus anderen Bereichen leicht verständlich und bieten zahlreiche weitere Anwendungsmöglichkeiten für das vorgestellte System.

1 Einführung

Die Inbetriebnahme von fertigungs- und verfahrenstechnischen Anlagen ist normalerweise mit hohem manuellem Aufwand im Bereich des Engineerings verbunden. Die wichtigste Aufgabe in diesem Bereich ist das Anlegen und Testen von Prozessführungsstrukturen für einzelne Komponenten der Anlage. In der Regel besteht ein Großteil dieser Aufgabe aus manuellem Kopieren der bewährten Bausteinstrukturen (z. B. in IEC 61131-3 Funktionsbausteinsprache (FBS) [Int13]), die nur leicht an die Anforderungen des aktuellen Projekts angepasst werden. Die auf diese Weise entstehenden Programme behalten die Trennung zwischen der Führungsstrategie und ihrer Implementierung oft nicht bei [KQE11].

Die Nachteile des manuellen Engineering-Workflows können durch den Einsatz regelbasierter Systeme überwunden werden, die zur Trennung der Führungsstrategie und deren Implementierung dienen. Regeln, die idealerweise unabhängig von der aktuellen Anwendung sind, konservieren das Engineering Know-how und lassen sich leicht übertragen und dokumentieren. Einige mögliche Einsatzszenarien eines regelbasierten Systems im Bereich des Anlagenengineering sind das automatische Erzeugen der Einzelsteuerebene ausgehend von R&I Fließbildern (Rohrleitungs- und Instrumentierungsfließbildern), die automatische Konsistenzüberwachung der Bilder und der Einzelsteuerebene und algorithmische Aggregation der existierenden Anlagendaten (z. B. Finden und Überwachen aktiver

Produktflusswege innerhalb verfahrenstechnischer Anlagen). Trotz beschriebener Vorteile bedarf ein regelbasiertes System an sich mehrerer sorgfältiger Designentscheidungen, die große Auswirkungen auf die Stabilität und die Akzeptanz des Systems haben. Eine zentrale Fragestellung ist dabei die Auswahl der Beschreibungssprache für den eingesetzten Regelkatalog.

In diesem Beitrag wird ein regelbasiertes System vorgestellt, welches deklarative Graphabfragen als Formalismus zur Regelbeschreibung nutzt. Das System setzt die graphbasierte NOSQL Datenbank Neo4J [RWE13] und deren Abfragesprache Cypher ein. Deren Syntax unterstützt im Vergleich zu anderen Abfragesprachen wie SQL (Structured Query Language) direkte Abfragen über Knoten und Kanten eines Graphs. Da die meisten Modelle aus der Prozessleittechnik wie R&I Fließbilder oder Code in FBS eine natürliche Darstellung als Graph besitzen, erscheint es sinnvoll Graphen bei der Konstruktion der Regeln einzusetzen. Außerdem ist die visuelle Syntax von Cypher sowohl für Ingenieure als auch für Informatiker verständlich. In diesem Beitrag werden wir einen Überblick über den Aufbau und die Einsatzszenarien des regelbasierten Systems verschaffen. Für detaillierte Anwendungsfälle und Details verweisen wir auf die Langversion dieses Beitrags [GWE14].

Der Beitrag ist wie folgt gegliedert: Abschnitt 2 stellt den Stand der Technik und die Anwendungen der NOSQL Datenbanken vor. Abschnitt 3.1 gibt eine kurze Einführung in Neo4J und Cypher. Abschnitte 3.2 und 3.3 bieten eine Übersicht über die Syntax und die Semantik der Regeln sowie über die Anbindung des regelbasierten Systems an diverse Datenquellen. Abschnitt 3.4 skizziert das regelbasierte Erzeugen von Kontrolllogik in FBS, welche auf R&I Fließbildern basiert. Im Abschnitt 4 folgt schließlich ein Ausblick auf zukünftige Anwendungsfelder des Systems und eine Zusammenfassung.

2 Stand der Technik

Der Einsatz von regelbasierten Systemen für Zwecke des Engineerings kann dem Forschungsbereich „Automatisierung der Automatisierung“ zugeordnet werden, dessen Ziel die systematische Vereinfachung des gesamten Automatisierungsprozesses ist [SSE09]. Automatisierung der Automatisierung ist ein recht weiter Bereich und umfasst u. A. Assistenzsysteme und die Mensch-Maschine Interaktion. Im Gegensatz dazu stellen regelbasierte Systeme nur eine Möglichkeit zur Speicherung und Wiederverwendbarkeit von Engineeringwissen dar [KQE11].

Wissensbasierte Systeme finden im Bereich der industriellen Automation bereits seit mehr als 25 Jahren Anwendung [KKY86]. Ein regelbasiertes System zum automatischen Erzeugen von Führungsstrukturen, bestehend aus PLT-Stellen und Interlocks, wurde in [SE06] vorgeschlagen und später auf den Bereich der Produktflusswegeverfolgung ausgedehnt [KQE11]. Diese Arbeiten setzen auf die imperative Darstellung der Regeln, die mithilfe von 61131 Sprachen definiert werden. Unser Ansatz unterscheidet sich daher nur durch ein anderes Regelbeschreibungsparadigma.

Die Nutzung generischer Abfragesprachen im Bereich der Automatisierung ist nicht neu. Zum Beispiel, wird XQuery in [LOML06] für die Abfragen auf XML-Daten genutzt. Im

Allgemein scheint XML als zentrales Format für den Austausch von Engineeringdaten geeignet zu sein [DB11]. Der von uns gewählte Einsatz lässt sich problemlos auf XML-Daten anwenden, für diesen Fall genügt es die als XML dargestellte Datenstruktur als einen Baumgraphen zu betrachten.

Generell finden NOSQL und insbesondere graphbasierte Datenbanken in unterschiedlichsten Anwendungsbereichen wie Bioinformatik [HJ13] oder Analyse sozialer Netzwerke [RWE13] Anwendung. Uns ist allerdings noch kein Beispiel aus dem Bereich der Automatisierung bekannt.

3 Regelbasiertes Engineeringssystem mit Graphabfragen

Vor der Präsentation des regelbasierten Systems möchten wir zunächst die graphbasierte Datenbank Neo4J [RWE13] und deren deklarative Abfragesprache Cypher vorstellen. Diese Technologien wurden zwar für die aktuelle Implementierung ausgewählt, beeinflussen aber das Gesamtkonzept der deklarativen Regeln nur minimal, sodass auch andere Abfragesprachen oder Datenbanken ins System eingebunden werden können.

3.1 Neo4J und Abfragesprache Cypher

Die Daten werden in Neo4J als Property-Graphen dargestellt. Diese Graphen sind einfache gerichtete Multigraphen, die aus Knoten und benannten Kanten, auch Relationen genannt, bestehen. Zusätzlich können sowohl die Knoten als auch die Kanten Eigenschaften (sogenannte Properties) beinhalten. Die Properties werden als Key-Value Paare gespeichert, wobei die Keys immer Strings sind und die Values von beliebigem Datentyp sein können. Ein einfacher Property-Graph ist in Abbildung 1 zu sehen. Der Graph stellt einen einfachen Kontrollzusammenhang in einem Durchflussregler dar. Knoten repräsentieren drei Mess- bzw. Steuerstellen für einen Durchflusssensor und eine Pumpe sowie einen Regler. Die Kanten vom Typ READS und SETS bilden den typischen Informationsfluss in einem Regler ab. Der Regler erhält den Messwert der Strecke vom Sensor und ändert aufgrund dieser Information die Vorgabe für den Aktor.

Die native Abfragesprache für Neo4J heißt Cypher. Sein Alleinstellungsmerkmal ist im Vergleich zu SPARQL oder Gremlin die deklarative Art der Abfragen mithilfe von ASCII-basierten Patterns. Die Syntax ist ähnlich zu der von SQL und ist auch für Nichtspezialisten intuitiv verständlich. Als Beispiel betrachten wir eine Abfrage in Listing 1. Die MATCH Klausel definiert das gesuchte strukturelle Muster. Demnach werden zwei als „u“ und „c“ bezeichnete Knoten gesucht, die durch eine gerichtete Kante „r“ verbunden

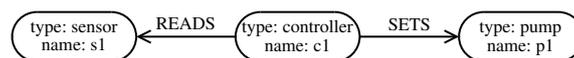


Abbildung 1: Ein einfacher Property-Graph

```

START u=node(*)
MATCH (u)<-[r]-(c)
WHERE c.type = "controller"
RETURN c.name, type(r), u.name as unit;

```

Listing 1: Einfache Abfrage in Cypher

```

+-----+-----+-----+
| c.name | type(r) | unit |
+-----+-----+-----+
| c1     | "READS" | s1   |
+-----+-----+-----+
| c1     | "SETS"  | a1   |
+-----+-----+-----+

```

Listing 2: Ergebniss der Abfragenausführung

sind. Die darauf folgende WHERE Klausel schränkt ähnlich wie in SQL die Auswahl der gefundenen Objekte weiter ein. So werden im Beispiel nur Knoten „c“, deren Eigenschaft „type“ „controller“ lautet, zugelassen. Die Auswertung dieser Abfrage auf dem Beispielgraphen aus Abbildung 1 liefert eine Ergebnistabelle aus Listing 2. Ähnlich wie bei SQL-Abfragen tragen die Spalten dieser Tabelle die Namen der Felder aus der RETURN Klausel der ausgeführten Abfrage. Die Spalten der Tabelle beinhalten alle mit der Abfrage übereinstimmenden Muster. Dieses einfache Beispiel reicht aus, um die Funktionsweise des regelbasierten Systems zu verstehen. Für Details über Cypher und Neo4J verweisen wir auf [RWE13].

Graphdatenbanken bieten mehrere Vorteile im Vergleich zu relationalen Datenbanken. Der wichtigste Vorteil von Neo4J ist die bessere Skalierbarkeit der Abfragen auf lokalen Eigenschaften der Graphen. Die lokalen Eigenschaften sind besonders für die Analyse stark vernetzter Graphen wie sozialer Netzwerke interessant. Zum Beispiel gehört das Abfragen der Anzahl der Nachbarn eines Knotens zu dieser Abfrageklasse [HJ13]. Bei Abfragen der Eigenschaften, die sehr viele Knoten eines Graphen besuchen müssen, sind hingegen etablierte relationale Datenbanken im Vorteil. Für die Zwecke des Engineerings sind aber die Unterschiede in der Leistung nur von nachrangiger Bedeutung. Wir sehen Potenziale vor allem in der Abfragesprache Cypher und deren intuitiver Syntax, die auch für Nichtexperten zugänglich ist.

3.2 Darstellung der Regeln

In diesem Abschnitt beschreiben wir den Aufbau der Regeln und deren Semantik. Eine Regel besteht aus einer Prämisse und einer Konklusion. Die Prämisse ist als Graphabfrage in Cypher formuliert und die Konklusion führt eine Reihe Operationen auf der Ergebnistabelle der Abfrage aus. Diese Semantik ist eine FORALL-DO Semantik, die anstatt der in [SE06, KQE11] eingeführten IF-THEN Semantik verwendet wird. Die FORALL-DO Semantik kann angewendet werden, da die Graphabfrage direkten Einfluss auf die Auswahl

```

<rule name="Example rule">
  <forall>
    <query language="Cypher">
      <![CDATA[
        START u=node(*)
        MATCH (u)-[r]-(c)
        WHERE c.type = "controller"
        RETURN c.name, type(r), u.name as unit
      ]]>
    </query>
    <op>System.out.println("Unit {unit} is connected
      to controller {c.name}.")</op>
  </forall>
</rule>

```

Listing 3: Regel mit der vorgestellten Graphabfrage

```

Unit s1 is connected to controller c1.
Unit a1 is connected to controller c1.

```

Listing 4: Ergebniss der Ausführung der Regel aus Listing 3

der betrachteten Daten hat. Die Regeln werden im XML-Format abgelegt, das an RuleML [BTW01] angelehnt ist. Ein allgemein akzeptiertes Format für die Darstellung der Regeln ist eine der Grundvoraussetzungen für die Automatisierung der Automatisierung [SSE09].

Eine Regel, die die bekannte Abfrage aus Listing 1 als Prämisse nutzt, ist in Listing 3 dargestellt. Als Erstes wird die Abfrage der Prämisse ausgewertet, die Auswertung liefert eine Tabelle mit Ergebnissen (vgl. Listing 2). Diese Tabelle bietet die Grundlage für die Parametrisierung der Operationen. Jede Operation der Regel wird auf jede Zeile der Tabelle angewendet. Die Werte der aktuellen Zeile können dabei als Argumente der Operationen fungieren. Dazu werden durch zwei geschweifte Klammer umschlossene Namen der Spalten mit dem aktuellen Datenwert ersetzt. Die Ausführung der Regel liefert die Ausgabe aus Listing 4, da die Operation Standardausgabefunktionen von Java kapselt.

Diese einfache Ausführungssemantik reicht aus, um komplexe Engineering-Operationen zu automatisieren. Die Mächtigkeit einer Regel hängt dabei von den zur Verfügbaren stehenden Daten und Operationen ab. Da unser System in Java implementiert ist, können wir die Reflexion ausnutzen, um beliebige Funktionen als Operationen einer Regelschlussfolgerung auszuführen. Das System ist somit sehr einfach zu erweitern.

In dem folgenden Abschnitt stellen wir die Architektur des regelbasierten Systems für die Analyse der Daten aus R&I Fließbildern und deren Anwendungsszenarien im Engineering vor.

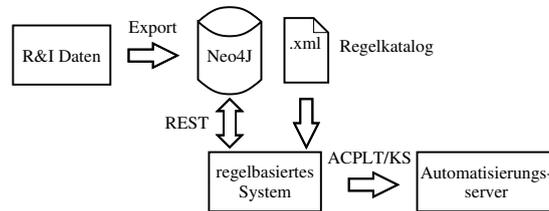


Abbildung 2: Übersicht der Systemarchitektur. Die Pfeile kennzeichnen Informationsflüsse

3.3 Datenexport und Verbindung zum Automatisierungsserver

Die Architektur der prototypischen Implementierung ist in Abbildung 2 dargestellt. R&I Fließbilder werden in eine Graphdarstellung umgewandelt und in der Datenbank gespeichert. Die Export-Anwendung ist an Neo4J über REST angebunden.

Nach der Überführung in die Graphdarstellung kann das regelbasierte System auf den R&I Informationen operieren, dazu werden Regeln aus einem Regelkatalog angewendet, die im XML-Format abgelegt ist. Die implementierten Operationen umfassen grundlegende Kommunikation mit ACPLT Automatisierungsservern über das offene Kommunikationsprotokoll ACPLT/KS [Alb03]. Mithilfe dieser ist es möglich IEC 61131-3 Funktionsbausteine und Datenverbindungen zur Laufzeit [GKE12] anzulegen.

Wegen der durchgängigen Nutzung der IP-basierten Kommunikation ist jede Systemkomponente aus Abbildung 2 über das Netzwerk verteilbar. Normalerweise unterliegt nur der Automatisierungsserver den harten Echtzeitbedingungen. Das Verarbeiten der Regeln darf somit weniger vorhersagbar bzw. performant sein und ein Tradeoff zwischen der Mächtigkeit der deklarativen Graphabfragen und der Dauer der Verarbeitung ist somit möglich.

3.4 Code-Generierung aus R&I Fließbildern

Für die Analyse von R&I Informationen wird die maschinenlesbare Darstellung im PandIX (Piping and Instrumentation Diagram Exchange) Format verwendet [SE12, ERD11]. PandIX bietet eine XML-Darstellung von R&I Fließbildern, die wiederum auf CAEX (Computer Aided Engineering Exchange) basiert [Int08].

Wie jedes XML-Dokument kann auch die PandIX Information als ein Baum interpretiert werden, dessen Knoten hierarchische Beziehungen zueinander haben. So kann z. B. eine Anlage-Teilanlage Beziehung zwischen zwei Anlagenteilen modelliert werden. Zusätzlich bietet PandIX die Möglichkeit Produkt- und Kontrollbeziehungen zwischen Anlagenkomponenten und PLT-Stellen herzustellen. Die Produktbeziehung modelliert die physische Verbindung zwischen zwei Komponenten z. B. mittels einer Rohrleitung. Die Kontrollbeziehung modelliert Verbindungen zwischen Anlagenkomponenten und PLT-Stellen wie im nächsten Paragraphen verdeutlicht.

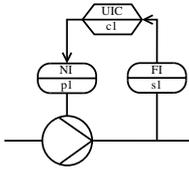


Abbildung 3: Beispiel eines R&I Fließbildes mit einer Flussregelung

Betrachten wir ein einfaches R&I Fließbild auf Abbildung 3. Dort sind eine Pumpe und eine bezüglich der Pumprichtung dahinterliegende Messstelle zu sehen. Die Pumpe und die Messstelle sind zu PLT-Stellen „pl“ bzw. „sl“ verbunden, die Schnittstellen zwischen der Physischen- und der Cyberwelt bilden. Die Signal- und Funktionscodes „NI“ und „FI“ deuten auf analoge Stell- bzw. Messgrößen der Motordrehzahl der Pumpe bzw. des Durchflusses am Sensor.

Die Interpretation der PLT-Stellen des Fließbildes als Property-Graphen liefert eine zu Abbildung 3 ähnliche Struktur. Die Typen der einzelnen PLT-Stellen und deren Verbindung zueinander sind beibehalten worden. Diese Informationen sind bereits ausreichend, um mit Graphabfragen, die ähnlich zu der Abfrage in Listing 1 aufgebaut sind, Grundkontrollbausteine für die Pumpe und den Durchflusssensor anlegen zu können (vorausgesetzt das Vorhandensein der entsprechenden Bausteintypen). Zusätzlich kann für die PLT-Stelle „c1“ ein Regler angelegt werden, welcher bereits mit den Ein- und Ausgängen der Kontrollbausteine verbunden ist. Sind zusätzlich Informationen über die I/O-Verbindungen der Pumpe und des Sensors bekannt (z. B. über die Auswertung einer Verschaltungsliste), so kann die Grundautomatisierung vollständig durch ein regelbasiertes System durchgeführt werden.

Diese Regeln können nicht nur für das Anlegen der Bausteine während der Anlagenplanung, sondern auch für ständige Konsistenzüberwachung zwischen den R&I Daten und den vorhandenen Führungsstrukturen verwendet werden. Dieses ist sogar mit der heutigen Implementierung möglich, falls die Automatisierung mithilfe von FBS erfolgt.

4 Zusammenfassung und Ausblick

In diesem Beitrag wurde ein Konzept eines regelbasierten Systems vorgestellt, das für die Automatisierung des Engineering-Prozesses eingesetzt werden kann. Die Besonderheit des Systems ist die Nutzung deklarativer Graphabfragen für das Formulieren einzelner Regeln. Die Nutzung von Graphabfragen bietet zwei Vorteile: Zum einen haben die meistverwendeten Modelle im Bereich der Prozessleittechnik wie R&I Fließbilder oder Kontrolllogik in FBS eine natürliche Graphdarstellung, zum anderen ist eine visuelle Darstellung der Abfragen für Mitarbeiter aus unterschiedlichen Bereichen zugänglich.

Im Abschnitt 3 wurde die Lösung einer alltäglichen Aufgabe aus dem Bereich des Engineerings dargestellt. Es handelt sich um das Anlegen der Bausteine für die Basisautomatisierung und der Regler ausgehend aus den Daten der R&I Fließbildern. Die tatsächlichen

Regeln, die dafür notwendig sind, sind in der Langfassung dieses Beitrages zu finden [GWE14]. Das automatische Erzeugen von Kontrollstrukturen beschränkt sich keinesfalls auf die in diesem Beitrag angesprochenen Bausteine der IEC 61131-3. Mithilfe vorgestellter Techniken können Elemente jeder anderen visuellen Programmiersprache wie z. B. MATLAB/Simulink erzeugt werden. Auch das Erzeugen textueller Programme ist möglich.

In der Langfassung [GWE14] wird ein weiteres Anwendungsszenario des regelbasierten Systems vorgestellt: das Erkunden und das Überwachen möglicher Produktflusswege innerhalb einer verfahrenstechnischen Anlage. Es stellte sich heraus, dass die Performance der Abfragen für diese Anwendung nicht ausreicht, da die Syntax von Cypher das Ausdrücken der Eigenschaften der Pfade in der WHERE Klausel (noch) nicht zulässt. Die Laufzeit der Abfragen der lokalen Eigenschaften des Graphs wie für die betrachtete Aufgabe der Codegenerierung ist hingegen sogar für Anwendungen im Produktivbetrieb unbedenklich.

Das Speichern des Know-hows in einer deskriptiven Art und Weise ist eine vielversprechende Alternative zu dem heutigen Engineering-Workflow. Die Fragestellungen der aktuellen Forschung umfassen das Erweitern des vorgestellten Regelwerks auf die Analyse und Modifikation der Kontrollstrukturen in FBS oder der Ablaufsprache zur Laufzeit. Das Konzept und die Notwendigkeit solcher Modifikation wurde in [GE13] vorgestellt. Eine weitere Forschungsrichtung ist das Anwenden deskriptiver Systeme auf weitere Aufgaben des klassischen Engineerings wie die Analyse von Verschaltungslisten oder das Anlegen von Interlock Schaltungen.

Literatur

- [Alb03] Harald Albrecht. On Meta-Modeling for Communication in Operational Process Control Engineering. *at-Automatisierungstechnik/Methoden und Anwendungen der Steuerungs-, Regelungs- und Informationstechnik*, 51(7/2003):339–340, 2003.
- [BTW01] Harold Boley, Said Tabet und Gerd Wagner. Design Rationale of RuleML: A Markup Language for Semantic Web Rules. Seiten 381–401, 2001.
- [DB11] R. Drath und M. Barth. Concept for interoperability between independent engineering tools of heterogeneous disciplines. In *IEEE 16th Conference on Emerging Technologies in Factory Automation (ETFA)*, Seiten 1–8, 2011.
- [ERD11] Ulrich Epple, Markus Remmel und Oliver Drumm. Modellbasiertes Format für RI-Informationen. *Automatisierungstechnische Praxis : atp*, 1-2, 2011.
- [GE13] Sten Grüner und Ulrich Epple. Paradigms for Unified Runtime Systems in Industrial Automation. In *Proceedings of the 12th European Control Conference (ECC): July 17-19, Zuerich, Switzerland*, Seiten 3925–3930, 2013.
- [GKE12] Sten Grüner, David Kampert und Ulrich Epple. A Model-Based Implementation of Function Block Diagram. In *Tagungsband Modellbasierte Entwicklung eingebetteter Systeme (MBEES)*, Seiten 81–90, März 2012.

- [GWE14] Sten Grüner, Peter Weber und Ulrich Epple. Rule-Based Engineering Using Declarative Database Queries [eingereicht]. In *INDIN 2014: IEEE 12th International Conference on Industrial Informatics*, 2014.
- [HJ13] Christian Theil Have und Lars Juhl Jensen. Are graph databases ready for bioinformatics? *Bioinformatics*, 29(24):3107, 2013.
- [Int08] International Electrotechnical Commission. *IEC 62424, Representation of process control engineering - Requests in P&I diagrams and data exchange between P&ID tools and PCE-CAE tools*. 2008.
- [Int13] International Electrotechnical Commission. *IEC 61131-3, Programmable controllers - Part 3: Programming languages*. 2013.
- [KKY86] Yasuhiro Kobayashi, Takashi Kiguchi und Toshiaki Yoshinaga. Development of Knowledge-based Method for Three-dimensional Pipe Route Planning. *Hitachi Rev.*, 35(1):13–16, Februar 1986.
- [KQE11] Tina Krausser, Gustavo Quirós und Ulrich Epple. An IEC-61131-based Rule System for Integrated Automation Engineering: Concept and Case Study. In *INDIN 2011: IEEE 9th International Conference on Industrial Informatics*, Lisbon, Juli 2011. IEEE.
- [LOML06] Omar J Lopez Orozco und Jose L Martinez Lastra. Using semantic web technologies to describe automation objects. *International Journal of Manufacturing Research*, 1(4):482–503, 2006.
- [RWE13] Ian Robinson, Jim Webber und Emil Eifrem. *Graph Databases*. O'Reilly Media, 2013.
- [SE06] Stefan Schmitz und Ulrich Epple. On Rule Based Automation of Automation. In Breitenecker F. and Troch I., Hrsg., *5th MATHMOD 2006: 5th Vienna Symposium on Mathematical Modelling*, 08. - 10.02.2006, Jgg. 30 of *AGRESIM Report*, Wien, Februar 2006. AGRESIM-Verlag.
- [SE12] A. Schüller und U. Epple. PandIX - Exchanging P&I diagram model data. In *IEEE 17th Conference on Emerging Technologies in Factory Automation (ETFA)*, Seiten 1–8, 2012.
- [SSE09] S. Schmitz, M. Schluetter und U. Epple. Automation of Automation – Definition, components and challenges. In *IEEE 14th Conference on Emerging Technologies in Factory Automation (ETFA)*, Seiten 1–7, 2009.

Assuring Standard Conformance of Partial Interfaces

Hardi Hungar

Institute of Transportation Systems
German Aerospace Center (DLR)
Braunschweig, Germany
hardi.hungar@dlr.de

Abstract: A current standardization effort for track-side equipment in German railways faces the difficulty of having to proceed incrementally. This means that only some of the interfaces of complex entities like interlocking controllers are specified while others remain under the control of the diverse manufacturers. As these other interfaces are necessary for testing the specified ones for standard conformance, a specific approach has to be devised to be able to achieve this goal. This paper presents the problem in its practical setting and the way it is intended to be solved.

1 Problem Statement

Notwithstanding the by now more than twenty years of efforts of standardizing railway control systems in Europe, proprietary interfaces and resulting incompatibilities between equipment components are still abundant. Any attempt at improving the situation faces the difficulty that the necessity of keeping the railway system in operation—defective equipment has to be replaced—and political demands—e.g. timeframes for new lines are set by third parties—that often a compromise between systematic and pragmatic solutions has to be found. Adding to this are the high costs of buying equipment and bringing it into operation.

An approach currently employed by German Railways is to incrementally specify the interface behavior of new equipment components. I.e., only one of the interfaces of an interlocking system is specified and shall be tested: the *focus interface*. The other interfaces remain to be considered in the future.

This simplifies the task of standardisation as not everything has to be done at once. But the downside is that this approach faces an inherent difficulty when it comes to testing. To drive the focus interface (and observe the correct interpretation of messages received over it), it is usually necessary to have access to (all the) other interfaces. Specification is easier by far—one can “internalize” the uncontrolled interfaces by subsuming everything in internal behavior of a specification automaton.

The left part of Fig. 1 shows a schematic view of a system. The ellipses mark external interfaces to neighboring systems. The focus interface on the left of the object is shown with some internal detail. The specification shall address the functional level of the *Rail*

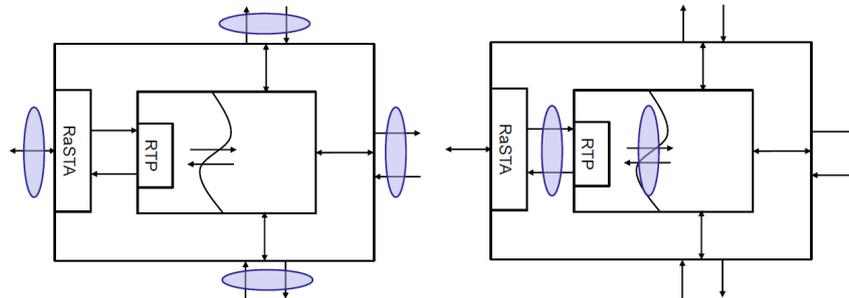


Figure 1: Schema of a system with four interfaces, of which one is to be specified (left), and the specification view of the system (right).

*Technical Protocol (RTP) and abstract from the concrete implementation of communication through the Rail Safe Transport Application (RaSTA), which utilizes an ethernet connection.*¹

The right part of Fig. 1 gives the specification view, where an additional virtual internal interface is added. In the specification, telegrams on the focus interface are related to messages and observations on this virtual internal one. Technically, these messages and observations are just actions of UML state machines which make up the specification. They *reflect* actions happening on the other (*masked*) interfaces, but are not formally related to them.

This works for the specification. Everything which happens on the focus interface of a real system can be judged as conformant to the specification: If there is some behavior on the virtual interface, the observations is ok. Otherwise, it is not.

Testing can of course not be done in terms of the internal specification interface but needs the real behavior on the masked interfaces. I.e., test cases and test execution have to take the view of the left side, while their derivation must refer to the right one. The problem is acerbated by the unavailability of a precise relation between internal and masked interfaces. In current practice, such a relation does not even exist: There are considerable differences between the masked interfaces (whose standardisation is yet to be initiated) in the different manufacturers' implementations of the devices, as already mentioned above.

Summarizing, the problem of testing the conformance of a partial interface of a system has to deal with both, the specification view and the real architecture. These are shown in a combined picture in Fig. 2.

¹The technical details of the differences between RaSTA and RTP are of course important in practice but will not be discussed in depth here.

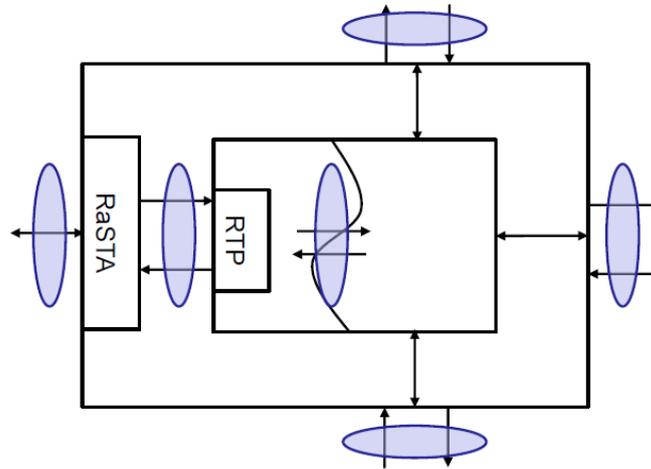


Figure 2: The combined testing and specification view.

2 Solution Approach

Differences between the implementations by different manufacturers call for integrating them into the test process in some way. Our solution relies on the assumed ability of the manufacturers of bridging the gap between (virtual) internal messages and commands and externals. The envisaged test architecture is depicted in Fig. 3.

The test rack adds two components to the test object:

Adapter internal-external: The manufacturer shall provide a module which translates between internal and masked interfaces. For its realization, interface drivers, simulators, or existing test interfaces accessing internals of the device may be used. Even a test engineer performing manual steps may be integrated via a suitable interface component.

Adapter RaSTA-RTP: This module must be provided by the test laboratory.

The test rack serves to provide the test object with an interface which is on the same level of abstraction as the specification. The remaining components of the test architecture are rather standard.

Test Execution Kernel: The kernel controls the test execution, i.e., it initializes the test objects, starts test sequences (including parameter completion in advanced scenarios), protocols the results, performs corrective actions (breaks and restarts if necessary) and generally monitors the execution. The kernel will be partially automatized.

Test Sequences: A data base with test sequences.

Test Report: A data base for detailed result data and accumulated reports.

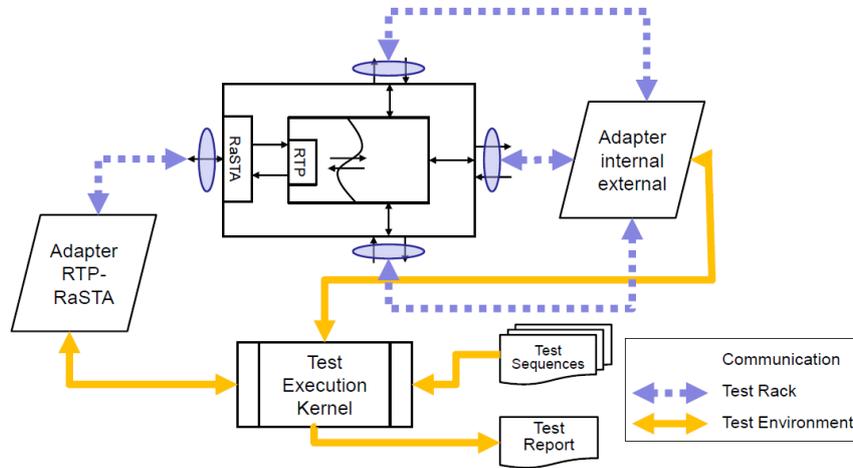


Figure 3: Components of the test architecture.

2.1 Validation

To be able to make qualified assertions of standard conformance, several arguments have to be spelled out. On the one hand, the correctness and completeness, resp. sufficient coverage, of the test cases wrt. the specification has to be checked. This involves techniques and methods from the domain of model based testing. Currently, manually derived test suites are evaluated for their suitability. In future enhancements of the overall approach, also test case generation from the specification models is intended to be considered.

Adapter design and validation will have to cope with the common problems of crossing abstraction levels (namely atomicity and timing issues as well as value concretizations). For the internal-external adapter a monitoring concept which observes its operation dynamically is envisioned. The user interface of an interlocking system provides many informations about internal states and thus qualifies as an adequate point of observation.

3 Conclusion

An approach to solve the problem of checking the standard conformance of complex rail devices wrt. partial interface specifications has been presented. The standardization concerns the functional interface aspect of the systems, including those real-time properties which are relevant to the systems' function (and safe behavior). The goal is to assert that system passing the test will be compatible in operation. The German Aerospace Center is involved in several ongoing activities which relate to the specific topic described here.