Tagungsband

Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme II

WS-Nr. 06022
Model-Based Development of Embedded Systems

09. – 13.01.2006

Informatik Bericht
TU Braunschweig
2006-01

Institut für
Software Systems Engineering
Technische Universität Braunschweig
Mühlenpfordtstraße 23
D-38106 Braunschweig

Inhaltsverzeichnis

Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme II

(WS-Nr. 06022: Model-Based Development of Embedded Systems)

Auch wenn ihr durch die Prägung des Begriffs „Model-Driven Architecture“ (MDA) durch die OMG zusätzliche Aufmerksamkeit zugekommen ist, hat die modellbasierte Entwicklung in vielen Bereichen von der Geschäftsprozessmodellierung bis hin zur Beschreibung von eingebetteten Steuerungssystemen Anwendung gefunden und geht dabei in den eingesetzten Techniken und Verfahren über die Trennung von plattformunabhängigen und plattformspezifischen Systembeschreibungen und den Übergang dazwischen hinaus.

Zentrales Merkmal der modellbasierten Entwicklung ist dabei der Einsatz von Modellen, die sich an der Problem- anstatt der Lösungsdomäne orientieren. Dies bedingt einerseits die Bereitstellung anwendungsorientierter Modelle (z.B. Matlab/Simulink-artige für regelungstechnische Problemstellungen, Statechart-artige für reaktive Anteile) und ihrer zugehörigen konzeptuellen (z.B. Komponenten, Signal, Nachrichten, Zustände) und semantischen Aspekte (z.B. synchroner Datenfluss, ereignisgesteuerte Kommunikation). Andererseits bedeutet dies auch die Abstimmung auf die jeweilige Entwicklungsphase, mit Modellen von der Anwendungsanalyse (z.B. Beispielszenarien, Schnittstellenmodelle) bis hin zur Implementierung (z.B. Bus- oder Task-Schedules, Implementierungstypen). Für eine durchgängige modellbasierte Entwicklung ist daher im Allgemeinen die Verwendung eines Modell nicht ausreichend, sondern der Einsatz einer Reihe von abgestimmten Modellen für Sichten und Abstraktionen des zu entwickelnden Systems (z.B. funktionale Architektur, logische Architektur, technische Architektur, Hardware-Architektur) nötig.

Durch den Einsatz problem- statt lösungszentrierter Modelle kann in jedem Entwicklungsabschnitt von unnötigen Festlegungen abstrahiert werden, während besonders wichtige und kritische Aspekte explizit und frühzeitig modelliert werden (z.B. Zeit, Prioritäten oder Kommunikationsaspekte). Die dadurch ermöglichte Anwendung analytischer und generativer Verfahren auf diesen Modellen ermöglicht die effiziente Entwicklung hochqualitativer Software.

Modellbasierte Vorgehensweisen haben gerade in der Softwareentwicklung in den letzten Jahren deutlich an Bedeutung gewonnen. Gerade im Bereich eingebetteter Software (z.B. Automotive oder Avionic Software Engineering) erfährt der Einsatz von domänenspezifischen Modellierungswerkzeugen in der Softwarenetwicklung zunehmend an Verbreitung. Wesentlich dazu haben dabei die Weiterentwicklung von Sprachen für aufgabenspezifische Modelle (z.B. synchroner Datenfluss) und dazugehörige Werkzeugen für spezialisierte Bereiche (z.B. Regelungs- und Steuerungsalgorithmen, Anlagensteuerung) und die Verbesserung der Ent-wicklungswerkzeuge, vor allem hinsichtlich Implementierungsqualität, Bedienkomfort und Analysemächtigkeit beigetragen.

Trotzdem sind im Kontext der modellbasierten Entwicklung noch viele, auch grundlegende Fragen offen, insbesondere im Zusammenhang mit der Durchgängigkeit. Die in diesen Tagungsband zusammengefassten Papiere stellen zum Teil gesicherte Ergebnisse, Work-In-Progress, industrielle Erfahrungen und innovative Ideen aus diesem Bereich zusammen und erreichen damit eine interessante Mischung theoretischer Grundlagen und praxisbezogener Anwendung.

Genau wie beim ersten, im Januar 2005 erfolgreich durchgeführten Workshop sind damit wesentliche Ziele dieses Workshops erreicht:

- 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)

Die Themengebiete, für die dieser Workshop gedacht ist und fachlich sehr gut abgedeckt sind, sich dieses Jahr (mit Ausnahmen) aber sehr stark auf den automotiven Bereich konzentrieren, fokussieren auf Teilaspekte modellbasierter Entwicklung eingebetteter Softwaresysteme. Darin enthalten sind unter anderem:

- Domänenspezifische Ansätze zur Modellierung von Systemen (z.B. Avionik, Railway, Automotive, Produktions- und Automatisierungstechnik)
- Durchgängigkeit und Integration von Modellen für eingebettete Systeme
- Modellierung spezifischer Eigenschaften eingebetteter Systeme (z.B. Echtzeiteigenschaften, Robustheit/Zuverlässigkeit, Ressourcenmodellierung)
- Konstruktiver Einsatz von Modellen (Generierung und Evolution)
- Modellbasierte Validierung und Verifikation

Das Organisationskomitee ist der Meinung, dass mit den Teilnehmern aus Industrie, Werkzeugherstellern und der Wissenschaft die bereits 2005 erfolgte Community-Bildung erfolgreich weitergeführt wurde, und damit demonstriert, dass eine solide Basis zur Weiterentwicklung des noch jungen Felds modellbasierter Entwicklung eingebetteter Systeme existiert.

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

Schloss Dagstuhl im Januar 2006,

Das Organisationskomitee

Holger Giese, Univ. Paderborn

Bernhard Rumpe, TU Braunschweig

Bernhard Schätz, TU München

Executive assistant

Holger Krahn, TU Braunschweig

Die ETAS Group, 2003 durch den Verbund der Unternehmen ETAS, Vetronix und LiveDevices entstanden, liefert ein komplettes Spektrum einheitlicher Entwicklungs- und Diagnosewerkzeuge, die den gesamten Lebenszyklus eines Steuergeräts umfassen. Die neue Verbindung der Entwicklungswerkzeuge mit den Diagnose- und Servicelösungen bildet die Basis für die Automotive LifeCycle Solutions-Strategie, mit der die ETAS Group die Vision von lebenszyklusübergreifenden Diagnosewerkzeugen im Fahrzeug Wirklichkeit werden lässt. Auf lange Sicht sieht sich die ETAS Group als starker Industriepartner, der Werkzeuge und Softwarekomponenten für die drei elementaren Aspekte elektronischer Steuergeräte anbietet (Control Functions, Diagnostic Functions und Platform Software Components).

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.

Das Institut für Software Systems Engineering (SSE) der TU Braunschweig entwickelt einen innovativen Ansatz des Model Engineering, bei dem ein Profil der UML entwickelt wird, das speziell zur Generierung modellbasierter Tests und zur evolutionären Weiterentwicklung auf Modellbasis geeignet ist (B. Rumpe: Agile Modellierung mit UML. Springer Verlag 2004). SSE ist auch Mitherausgeber des Journals on Software and Systems Modeling.

Der Lehrstuhl für Software Systems Engineering der TU München entwickelt in enger Kooperation mit industriellen Partnern modellbasierte Ansätze zur Entwicklung eingebetteter Software. Schwerpunkte sind dabei die Integration ereignisgetriebener und zeitgetriebener Systemanteile, die Berücksichtigung sicherheitskritischer Aspekte, modellbasierte Testfallgenerierung und modellbasierte Anforderungsanalyse, sowie den werkzeuggestützten Entwurf.

dSPACE ist der weltweit führende Anbieter von Lösungen für die Entwicklung und den Test schneller mechatronischer Regelungssysteme. Anwendungsbereiche für dSPACE-Systeme finden sich vor allem in der Automobilindustrie, aber auch in der Antriebstechnik, der Luft- und Raumfahrt und der Industrieautomation.

Das Software Quality Lab (s-lab) ist ein Institut für Kompetenz- und Technologietransfer, in dem Partner aus der industriellen Softwareentwicklung mit Forschungsgruppen der Universität Paderborn auf dem Gebiet der Softwaretechnik eng zusammenarbeiten. Zielsetzung des s-lab ist die Entwicklung und Evaluierung von konstruktiven und analytischen Methoden sowie Werkzeugen der Softwaretechnik, um qualitativ hochwertige Softwareprodukte zu erhalten. Eine hohe Relevanz für die industrielle Softwareentwicklung sowie die Notwendigkeit des Einsatzes wissenschaftlicher Methoden kennzeichnen die im s-lab bearbeiteten Fragestellungen.

Schloss Dagstuhl wurde 1760 von dem damals regierenden Fürsten Graf Anton von Öttingen-Soetern-Hohenbaldern erbaut. Nach der französischen Revolution und der Besetzung durch die Franzosen 1794 war Dagstuhl vorübergehend im Besitz eines Hüttenwerkes in Lothringen. 1806 wurde das Schloss mit den zugehörigen Ländereien von dem französischen Baron Wilhelm de Lasalle von Louisenthal erworben. 1959 starb der Familienstamm der Lasalle von Louisenthal in Dagstuhl mit dem Tod des letzten Barons Theodor aus. Das Schloss wurde anschließend von den Franziskus-Schwestern übernommen, die dort ein Altenheim errichteten. 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.

Eignung der UML 2.0 zur Entwicklung von Bordnetzarchitekturen

Michael von der Beeck

E/E Prozessentwicklung
BMW Group
80788 München
Michael.Beeck@bmw.de

Abstract: In diesem Positionspapier wird die Eignung der UML 2.0 zur Modellierung von Bordnetzarchitekturen in der Automobilindustrie untersucht. Diese Untersuchung wird durch eine Gegenüberstellung relevanter Charakteristika von Fahrzeug-IT und Business-IT motiviert. Ein potentielles Problem bei dem Einsatz der UML 2.0 wird bei einer Betrachtung der Architektur- und Komponentenentwicklung in der Fahrzeug-IT sichtbar. Hieraus wird als Handlungsbedarf die Überprüfung der Integrierbarkeit der UML 2.0 mit etablierten, domänen-spezifischen Modellierungsnotationen abgeleitet.

1 Einleitung

In diesem Positionspapier wird die Eignung der UML 2.0 zur Erstellung von Architekturen im Automotive-Kontext untersucht. Hierzu wird zunächst ein schematischer Überblick über die Architektur- und Komponentenentwicklung für Fahrzeugbordnetze gegeben. Dann werden Charakteristika der beiden Bereiche Fahrzeug-IT und Business-IT der Automobilindustrie gegenübergestellt. Anschließend werden Eigenschaften der UML 2.0 aufgeführt, die für bzw. gegen ihren Einsatz in der Fahrzeug-IT sprechen. Nachfolgend wird der Aspekt der Integration von Architektur- und Komponentenentwicklung in der Fahrzeug-IT betrachtet. Hierbei steht eine Integration von Architekturmodellen, die mit UML 2.0 erstellt werden, und von Komponentenmodellen, die mit etablierten, domänen-spezifischen Modellierungsnotationen wie MATLAB/Simulink oder ASCET erstellt werden, im Vordergrund. Als Fazit resultiert ein Handlungsbedarf: Die Integrierbarkeit von UML 2.0-Modellen und MATLAB/Simulink- bzw. ASCET-Modellen soll überprüft werden.

2 Architekturentwicklung für Fahrzeugbordnetze

Die nachfolgende Abbildung zeigt schematisch einzelne Schritte bei der Entwicklung von Fahrzeugbordnetzen. Dabei werden lediglich die Architektur-Entwicklung und die Komponenten-Entwicklung relativ detailliert betrachtet.

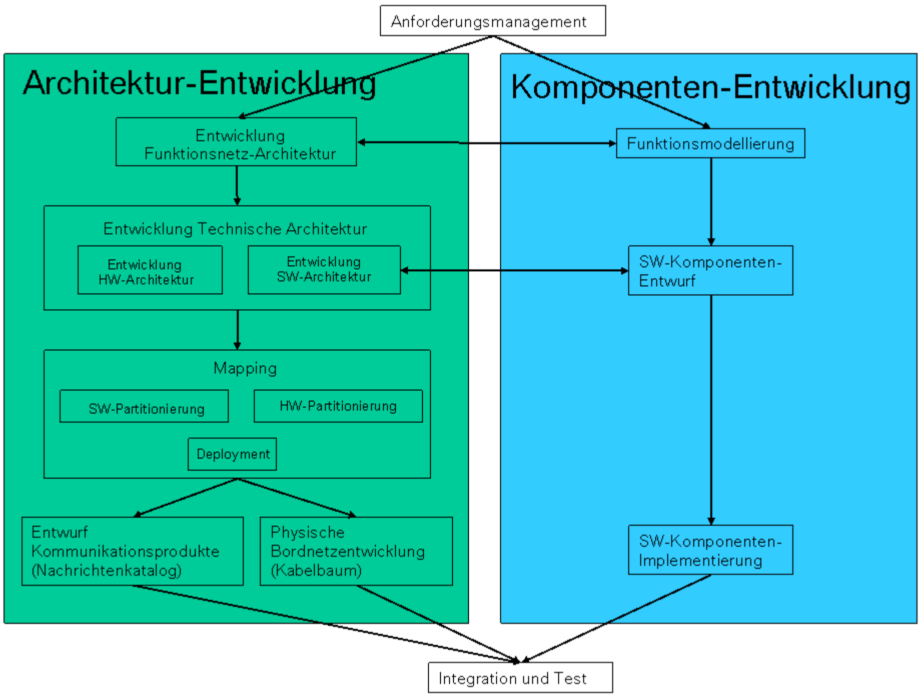

Anforderungsmanagement
Architektur-Entwicklung
Komponenten-Entwicklung
Entwicklung Funktionsnetz-Architektur
Funktionsmodellierung
Entwicklung Technische Architektur
Entwicklung HW-Architektur
Entwicklung SW-Architektur
SW-Komponenten-Entwurf
Mapping
SW-Partitionierung
HW-Partitionierung
Deployment
Entwurf Kommunikationsprodukte (Nachrichtenkatalog)
Physische Bordnetzentwicklung (Kabelbaum)
SW-Komponenten-Implementierung
Integration und Test

Abbildung 1: Architektur-Entwicklung und Komponenten-Entwicklung für Fahrzeugbordnetze

Zum Verständnis der Abbildung 1 sollen folgende Begriffsklärungen dienen:

- Funktionsnetz-Architektur:
  Diese stellt gesamthaft, aber implementierungsunabhängig, die Menge der zu realisierenden Funktionen eines Bordnetzes dar. Zusätzlich ist die Kommunikation zwischen den einzelnen Funktionen grob beschrieben.

- SW-Architektur:
  Diese stellt gesamthaft die SW-Komponenten zur Realisierung der Bordnetzfunktionen sowie ihre wechselseitige Kommunikation dar.

- HW-Architektur:
  Diese stellt die Menge der Steuergeräte (= ECUs), der Busverbindungen zwischen den Steuergeräten und der Gateways (zur Datenübertragung zwischen verschiedenen Bussen) dar.

- SW-Partitionierung:
  Dies ist eine Abbildung der Funktionsnetz-Architekur auf die SW-Architektur.

- HW-Partitionierung:
  Dies ist eine Abbildung der Funktionsnetz-Architektur auf die HW-Architektur.

- Deployment:
  Dies ist eine Abbildung der SW-Architektur auf die HW-Architektur.

Die linke Seite der Abbildung 1 ist durch die Architektur-Entwicklung gegeben. Die einzelnen Architekturen (Funktionsnetz-Architektur, HW-Architektur und SW-Architektur), die während der Architektur-Entwicklung entstehen, stellen jeweils Systemsichten dar, d.h. gesamthafte Darstellungen des zu erstellenden Systems. Bei der Entwicklung von Bordnetzen steigt die Notwendigkeit zur Verwendung von Systemsichten aufgrund der starken Zunahme der Komplexität von Bordnetzen.

In der Automobilindustrie hat sich innerhalb der Architekturentwicklung bisher die Systemsicht HW-Architektur etabliert (als Synonyme hierfür werden vielfach die Begriffe Steuergerätenetzwerk oder Bordnetztopologie verwendet), dahingegen müssen sich die Systemsichten Funktionsnetz-Architektur und SW-Architektur erst noch etablieren.

3 Gegenüberstellung von Fahrzeug-IT und Business-IT im Automotive-Bereich

Im Automotive-Bereich lässt sich die Informationstechnologie (IT) in die beiden Bestandteile Business-IT und Fahrzeug-IT zerlegen.

3.1 Charakterisierung der Business-IT

Beispiele für Bestandteile der Business-IT im Automotive-Bereich sind:

- Händler-Applikationen (z.B. Diagnose-Software)

- Produktionsplanungs-Software

- (fahrzeug-externe) Datenbanken für Autonavigation

Die Verwendung von Systemsichten ist in der Business-IT etabliert.

Es existiert nur eine geringe Anzahl eingebetteter Systeme in der Business-IT.

In der Business-IT ist der Einsatz der UML zur Modellierung weit verbreitet – sie konstituiert hier bereits einen de-facto-Standard.

3.2 Charakterisierung der Fahrzeug-IT

Steuergeräte (z.B. Anti-Blockiersystem, Motorsteuerung, On-board Navigation) sind typische Bestandteile der Fahrzeug-IT.

In der Fahrzeug-IT ist die Verwendung von Systemsichten für die “späte Phase” der Architekturentwicklung (konkret: für die Entwicklung der HW-Architektur) etabliert, jedoch nicht für die “frühe Phase” der Architekturentwicklung (konkret: für die Entwicklung der Funktionsnetz-Architektur).

Es existiert eine große Anzahl eingebetteter Systeme in der Fahrzeug-IT, insbesondere stellen die oben erwähnten Steuergeräte meist eingebettete Systeme dar. Ihre Entwicklung erfordert die Berücksichtigung charakteristischer Anforderungen – hierzu gehören z.B. Anforderungen durch limitierte HW-Ressourcen, Aufstartzeiten, Energieverbrauch und Realzeit. Zur Modellierung und Implementierung einzelner Komponenten (konkret: einzelner Steuergeräte) der Fahrzeug-IT (vgl. „Komponenten-Entwicklung“ in Abbildung 1) werden fast ausschließlich domänen-spezifische Notationen/Tools wie MATLAB/Simulink und ASCET eingesetzt, jedoch kaum die UML.

4 Eigenschaften der UML 2.0

In den beiden nachfolgenden Unterkapiteln werden Vor- und Nachteile der UML 2.0 hinsichtlich ihrer Eignung zur Architekturmodellierung aufgeführt.

4.1 Vorteile der UML 2.0

Die UML 2.0 ist sehr ausdrucksmächtig.

Die UML ist sehr weit verbreitet.

Es existiert bereits eine umfassende Toolunterstützung der UML 2.0 durch viele Toolhersteller.

Die UML 2.0 erlaubt eine leichte Modellierbarkeit von Partitionierung und Deployment durch Verwendung ihrer Verteilungsdiagramme, die genau für diese Aufgabe definiert wurden.

Die UML 2.0 bietet eine systematische Möglichkeit zur Sprachanpassung durch Definition von (domänen-spezifischen) Profilen. Beispiele für UML 2.0-Profile sind:

- SysML: Systems Engineering-spezifische Anpassung der UML 2.0
- EAST-ADL: automotive-spezifische Architekturbeschreibungssprache auf UML 2.0-Basis

4.2 Nachteile der UML 2.0

Die UML wird in der Fahrzeug-IT kaum zur Komponentenentwicklung eingesetzt. Entwickler von Funktionsmodellen und von Software für Steuergeräte verwenden stattdessen domänen-spezifische Notationen/Tools wie MATLAB/Simulink und ASCET.

Die UML ist keine domänen-spezifische Sprache, insbesondere keine spezifische Sprache zur Entwicklung eingebetteter Systeme.

Die UML ist (lediglich) eine Notation, jedoch keine Methode: Ihr fehlen also Regeln zur konkreten (Architektur-)Modellierung.

Eine Folgerung aus den beiden letzten Aussagen lautet: Für einen erfolgreichen Einsatz der UML 2.0 ist ein domänen-spezifisches Regelwerk erforderlich. Konkret müssten daher UML 2.0-Modellierungsrichtlinien zur Architekturentwicklung von Bordnetzen erstellt werden.

5 Integration von Architektur- und Komponentenentwicklung in der Fahrzeug-IT

Eine Verwendung der UML 2.0 in der Architekturentwicklung der Fahrzeug-IT erscheint viel versprechend, da sich die UML generell zur Architekturbeschreibung bewährt hat (z.B. in der Business-IT).

Darüber hinaus wäre eine Verwendung der UML 2.0 in der Komponentenentwicklung der Fahrzeug-IT natürlich auch denkbar, da die UML außerhalb der Fahrzeug-IT bereits einen de-facto-Modellierungsstandard darstellt. Diese Verwendung würde aber kurzfristig einen sehr großen Änderungsaufwand in der Automobilindustrie (bei OEMs und Zulieferern) bedeuten und wird daher in diesem Positionspapier nicht weiter betrachtet.

Als Konsequenz ergibt sich, dass für den Einsatz der UML 2.0 zur Architekturmodellierung in der Fahrzeug-IT eine Integration zwischen den betroffenen Modellen der Architekturentwicklung und denen der Komponentenentwicklung realisiert werden muss, also (gemäß Abbildung 1) sowohl zwischen Funktionsnetzarchitektur und Funktionsmodellen als auch zwischen SW-Architektur und SW-Komponenten.[1]

Aspekte dieser Integration beziehen sich sowohl auf die Definition des Gesamtentwicklungsprozesses als auch auf die Integration der jeweiligen Modelle auf Ebene der verwendeten Modellierungssprachen (d.h. UML 2.0 einerseits und MATLAB/Simulink und/oder ASCET andererseits).

[1] Darüber hinaus ist natürlich die konkrete Verwendung der UML 2.0 zur Architekturmodellierung (am besten in Form von UML 2.0-Modellierungsrichtlinien) festzulegen.

Daher lässt sich der folgende primäre Handlungsbedarf ableiten:

Es besteht die Notwendigkeit zur Überprüfung der Integrierbarkeit von UML 2.0 mit etablierten Modellierungssprachen zur Komponentenentwicklung, also z.B. mit MATLAB/Simulink oder ASCET. Zu berücksichtigen ist hierbei, dass in Architekturmodellen (Funktionsnetz-Architekturen und SW-Architekturen) Strukturinformation vorherrscht, wohingegen Verhaltensinformation lediglich rudimentär vorhanden ist. Diese Einschränkung sollte die Integration erheblich erleichtern.

6 Zusammenfassung

Dieses Positionspapier motiviert den Einsatz der UML 2.0 zur Architekturmodellierung bei der Bordnetzentwicklung der Automobilindustrie.

Hierzu wird ein Handlungsbedarf formuliert: Die Integrierbarkeit von UML 2.0-Architekturmodellen (konkret: Funktionsnetz-Architektur- und SW-Architekturmodelle) mit Komponentenmodellen, die unter Verwendung domänenspezifischer Modellierungswerkzeuge wie MATLAB/Simulink und ASCET erstellt worden sind, muss überprüft werden.

Einsatz von Modell-basierten Entwicklungstechniken in sicherheitsrelevanten Anwendungen: Herausforderungen und Lösungsansätze[1]

Mirko Conrad, Heiko Dörr

DaimlerChrysler AG
Research E/E and Information Technology
Alt-Moabit 96A
10559 Berlin
mirko.conrad@daimlerchrysler.com
heiko.doerr@ daimlerchrysler.com

Abstract: Modell-basierte Entwicklungstechniken werden zunehmend auch in sicherheitsrelevanten Anwendungen eingesetzt. In derartigen Anwendungsszenarien müssen die Anforderungen von Normen und Standards aus dem Sicherheitsbereich adaptiert und auf die Modell-basierte Entwicklung abgebildet werden.

Der vorliegende Beitrag diskutiert dabei auftretende Herausforderungen und zeigt Lösungsmöglichkeiten auf. Der Schwerpunkt liegt hierbei auf fehlervermeidenden Maßnahmen.

1 Motivation und Einleitung

Die Modell-basierte Entwicklung hat sich in wichtigen automotiven Anwendungsbereichen als Standardparagdigma für die Steuergeräte-Softwareentwicklung etabliert [CFG+05]. In der Modell-basierten Entwicklung werden Modellierungs- und Simulationsumgebungen (*Modeling Packages*) wie MATLAB/ Simulink/Stateflow und darauf basierende Codegeneratoren wie der RealTime-Workshop/Embedded Coder oder TargetLink für die Entwicklungsphasen Detailed Design und Coding verwendet.

Anwender berichten übereinstimmend von erreichbaren Effizienzsteigerungen zwischen 20 und 50% durch den Übergang zu Modell-basierter Entwicklung mit Codegenerierung. Darüber hinaus wird ein schnellerer Anstieg des Reifegrads der entwickelten Funktionen erreicht.

Von diesen Vorteilen soll zukünftig auch im Bereich sicherheitsrelevanter Anwendungen profitiert werden können. In derartigen Anwendungsszenarien müssen jedoch zusätzliche Anforderungen von Normen und Standards aus dem Sicherheitsbereich adaptiert und - da die zumeist aus der Zeit vor Einführung der

[1] Die zugrunde liegenden Arbeiten wurden z.T. im Rahmen des BMBF Vorhabens IMMOS (01ISC31D) durchgeführt, siehe www.immos-project.de

Modell-basierten Entwicklung stammen - auf die Modell-basierte Entwicklung abgebildet werden.

Der vorliegende Beitrag diskutiert dabei - aufbauend auf [DC05] - auftretende Herausforderungen und zeigt Lösungsmöglichkeiten, die auf konkreten Projekterfahrungen basieren, auf.

Kapitel 2 listet relevante Standards und Richtlinien auf. Kapitel 3 beschreibt einige der existierenden Herausforderungen und zeigt Lösungsmöglich-keiten auf. Kapitel 4 fasst den Beitrag zusammen.

2 Normen, Standards und Guidelines

Als Informationsquellen für den Einsatz Modell-basierter Entwicklungstechniken in sicherheitsrelevanten Anwendungen können u.a. die nachfolgend aufgelisteten Normen, Standards und Guidelines dienen:

2.1 ISO TR 15497:2000 Road vehicles - Development guidelines for vehicle based software

Die Mitte der 1990er Jahre von der MISRA[2] erarbeiteten und dann in einen Technical Report der ISO überführten Development Guidelines for Vehicle Based Software stellen einen der ersten Standardisierungsversuche in Bezug auf die Softwareentwicklung im Automobilbereich dar. Eine Überarbeitung und Aktualisierung ist für die Zeit nach Abschluss der Arbeiten an der ISO 26262 angedacht.

2.2 MISRA-C:2004 Guidelines for the use of the C language in critical systems

Die ebenfalls von der MISRA erarbeiteten Guidelines for the use of the C language in critical systems beschreiben ein Subset der Programmiersprache C, dass für den Einstz in sicherheitskritischen Anwendungen geeignet ist

2.3 IEC61508:1998 Functional safety of electrical/electronic/programmable electronic safety-related systems

Die IEC 61508 ist eine anwendungsbereichsunabhängige (generische) Grundnorm für elektrische/elektronische/programmierbare elektronische Sicherheitssysteme (E/E/PES), die die Entwicklung sektorspezifischer Normen für E/E/PES erleichtern soll und übergangsweise bei der Entwicklung von E/E/PES in den Bereichen Anwendung finden, in denen es noch keine domainspezifische Norm gibt. Teil 3 der Norm beschäftigt sich mit Anforderungen an die Softwareentwicklung.

[2] The Motor Industry Software Reliability Association

2.4 ISO/WD 26262:2005 Road vehicles - Functional safety

Im Rahmen des FAKRA[3] AK 16 Funktionssicherheit wurde der Entwurf einer sektorspezifischen Ausprägung der IEC61508 für den Automobilbereich erarbeitet ('Automotive 61508'). Diese wurde Anfang November 2005 als Working Draft an die ISO mit dem Ziel der Standardisierung übergeben. ISO/WD 26262 enthält u.a. Teile zur Softwareentwicklung sowie zu übergreifenden Aktivitäten wie der Zertifizierung von Entwicklungswerkzeugen.

2.5 RTCA DO-178B/ED-12B Software Considerations in Airborne Systems and Equipment Certification

Die RTCA DO-178B bzw. ihr europäisches Gegenstück EUROCAE ED-12B definieren Richtlinien für die Entwicklung von Luftfahrt-Software. Die DO-178B hat sich seit ihrem Erscheinen zu einem De-facto-Standard für die Zertifizierung neuer Luftfahrt-Software entwickelt.

Im Rahmen der SC-205/WG-71 wird derzeit ein Nachfolgestandard DO 178-C erarbeitet, der im Dezember 2008 fertig gestellt werden soll. Ein Themenfeld im Rahmen der Überarbeitung stellt die Berücksichtigung der Modell-basierten Entwicklung dar.

3 Herausforderungen und Lösungsszenarien

3.1 Sprachbeschreibung der Modellierungssprache

Eine Grundvoraussetzung für den Einsatz einer Sprache im Rahmen sicherheitsrelevanter Projekte sollte eine allgemeine Sprachdefinition hinsichtlich Syntax und Semantik sein.

Während es für prozedurale Programmiersprachen wie C allgemein zugängliche Sprachstandards gibt (vgl. ISO/IEC 9899:1999), ist dies für die häufig verwendete Modellierungssprache Simulink/Stateflow nur in Ansätzen der Fall.

Syntax und Semantik der Modellierungssprache SL/SF sind in den Handbüchern zur Modellierungs- und Simulationsumgebung zwar umfänglich beschrieben, der Sprachbeschreibung fehlt es jedoch an Rigorosität. Die Semantik von Bussen und ähnlichen Konstrukten ist komplex und könnte besser dokumentiert sein. Zudem bleibt unklar, ob die Auflistung der Sprachelemente vollständig ist. Hinweise auf mehrdeutige oder fehleranfällige Sprachmittel muss sich der Nutzer anderweitig verschaffen. Nachteilig wirkt sich darüber hinaus aus, dass die Semantik einiger Sprachbestandteile über die verschiedenen Versionen von ML/SL/SF hinweg Modifikationen erfahren hat.

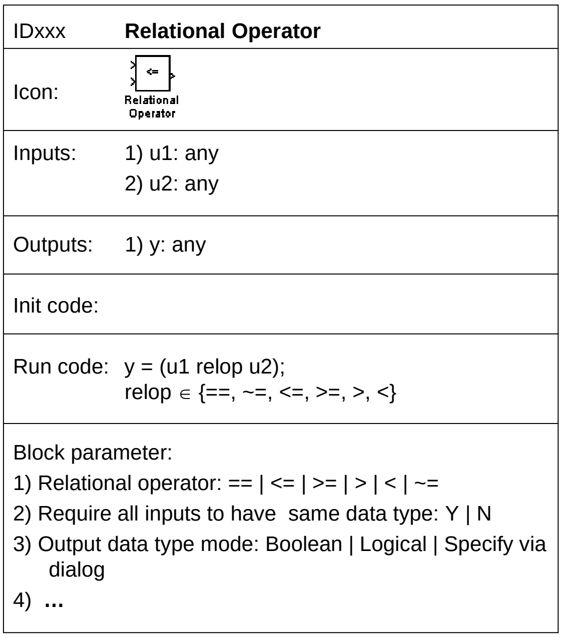
[3] Fachauschuß Kraftfahrzeuge

Die Semantik ist teilweise layoutabhängig. Semantikbeschreibungen tragen häufig exemplarischen Charakter.

In der Sekundärliteratur finden sich einige Aufsätze, die sich ausführlicher mit der Semantik von Simulink und Stateflow befassen [ASK04, CCM+03, SSC+04,].

Da eine einfach zu handhabende Semantik für die Gesamtsprache, insbesondere für ihren Zustandsautomatenanteil Stateflow, nicht zu erwarten ist, erscheint ein mehrstufiges, pragmatisches Vorgehen ratsam.

In einer ersten Stufe sollte zunächst eine vollständige Liste aller Simulink-Basisblöcke, Stateflow-Konstrukte und Verbindungselemente mit ihrer syntaktischen Varianz (z.B. Blockparameter) erstellt werden, die dann nachfolgend bezüglich ihrer Semantik beschreiben werden.



IDxxx	**Relational Operator**
Icon:	<= Relational Operator
Inputs:	1) u1: any 2) u2: any
Outputs:	1) y: any
Init code:	

Run code:	y = (u1 relop u2);
	relop ∈ {==, ~=, <=, >=, >, <}

Block parameter: 1) Relational operator: == | <= | >= | > | < | ~= 2) Require all inputs to have same data type: Y | N 3) Output data type mode: Boolean | Logical | Specify via dialog 4) **…**	

Abbildung 1: Exemplarische Dokumentation von Simulink-Basisblöcken

Eine pragmatische Beschreibung der Simulink-Basisblöcke könnte bspw. auf der von der MSR-MEGMA [MSR02] verwendeten Art und Weise basieren (siehe Abb. 1). Die Vielzahl der Blockparameter und ihre Kombination erschwert allerdings die Semantikbeschreibung, die zudem von Gesamtmodell- und Simulationseigenschaften abhängen kann.

Die vollständige Auflistung der syntaktischen Sprachelemente würde die Entwicklung von Prüf- und Unterstützungswerkzeugen erleichtern. Hilfreich wäre zudem die Bereitstellung der Grammatik für die .mdl-Files.

In einer weiteren Ausbaustufe wäre das Zusammenwirken der einzelnen Sprachelemente bei der Kombination zu Modellen und deren Ausführung zu beschreiben.

3.2 Sicheres Sprachsubset / Modellierungsrichtlinien

Bei der Modellierung funktionssicherer Anwendungen sind im Vergleich zu unkritischen Funktionsanteilen zusätzliche Randbedingungen zu beachten. Diese müssen für den Entwickler in geeigneter Form verfügbar gemacht werden. Nicht alle in der Modellierungssprache enthaltenen Konstrukte sind auch für sicherheitsrelevante Anwendungen sinnvoll.

Auf Basis der vollständigen Beschreibung der verwendeten Modellierungssprache (vgl. 3.1) ist zunächst eine für sicherheitsrelevante Anwendungen geeignete Sprachteilmenge abzuleiten. Ziel eines solchen *Safe Modeling Language Subsets* ist es, in sicherheitsrelevanten Anwendungen zum einen Sprachfeatures zu vermeiden,

- □ die nicht vollständig definiert sind, mehrdeutig oder layoutabhängig sind bzw. die zu Missverständnissen beim Benutzer führen können,
- □ die von verschiedenen Codegeneratoren in unterschiedlicher Weise umgesetzt werden können,
- □ die von verschiedenen Modellierungs- und Simulationswerkzeugen bei vergleichbarer Syntax unterschiedliche Semantiken haben

und zum anderen Workarounds bereitzustellen, um bekannte Probleme im verwendeten Modellierungs- und Simulationswerkzeug zu umgehen.

Das sichere Sprachsubset kann zunächst über eine Beschränkung der für sicherheitsrelevante Anwendungen erlaubten Modellelemente erreicht werden. Dies wird in praxi typischerweise über Positivlisten erlaubter Blöcke erreicht, die in eigenen Blockbibliotheken zusammengefasst werden. Darauf aufbauend können den Regeln formuliert werden, die zu unerlaubte Kombinationen und Einstellungen von Sprachelementen betreffen (*Safety Rules*).

Die in Abb. 2 (oben) gezeigte Regel beschränkt das zulässige Blockset für Fixedpoint-Modelle, die als Ausgangspunkt für die Codegenerierung sicherheitskritischer Software dienen, auf Multiplikationsblöcke mit zwei 2 Operanden.

Abb. 2 (unten) zeigt eine Safety-Rule für die Modellierung, die durch Übertragung einer ähnlichen Regel aus dem MISRA-C Sprachstandard entstanden ist. Hierbei sollen unerwartete Programmergebnisse als Folge von Vergleichen von Floatingpoint-Variablen auf exakte Gleichheit vermieden werden.

⇨ TC_0623 TL:
Vermeidung von Mehrfachmultiplikationen

Category:

☐ Readabilty	☐ Workflow	☐ V&V	☑ Code Gen.	☑ Safety

Automation:
possible

Description:
Bei Fixed Point-Modellen sollten bei ‚Product' nie mehr als zwei Signale miteinander verknüpft werden.

⇨ MC_002:
Absence of equality / inequality tests in floating-point expressions

Category:

☐ Readabilty	☐ Workflow	☐ V&V	☐ Code Gen.	☑ Safety

Automation:
possible

Description:
Floating-point expressions shall not be tested for equality or inequality.
The inherent nature of floating-point types is such that comparisons of equality will often not evaluate to true even when they are expected to.

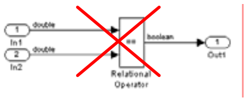


Fig 1-1: Comparison of floating point expressions for equality in Simulink

Abbildung 2: Definition eines Safe Modeling Language Subsets mit Hilfe von Modellierungsregeln

In Analogie zu einer geeigneten Teilmenge der Modellierungssprache für sicherheitskritische Anwendungen kann eine ähnliche Beschränkung auf der Ebene des daraus erzeugten Codes vorgenommen werden.

Ziele der Definition eines *Safe Implementation Language Subsets* sind die Vermeidung von möglichen Problemen in der Implementierungssprache, z.B

- von C Features, die nicht vollständig definiert oder mehrdeutig sind, bzw. das Potential für Missverständnisse beim Anwender bieten,
- von C Features, die durch verschiedene Compiler unterschiedlich gehandhabt werden können

sowie die Bereitstellung von Workarounds für bekannte Compilerprobleme.

Die genaue Definition einer sicheren C Sprachteilmenge ist davon abhängig, ob der C Code manuell oder automatisch erzeugt wird. Aus diesem Grunde wird zurzeit neben den weithin anerkannten MISRA-C Rules [MISRA04] im Rahmen des MISRA Autocode-Forums an einem Sprachsubset für Autocode gearbeitet.

Da derzeit die Erfahrungen mit Sprachbeschränkungen auf Implementierungsebene noch überwiegen, wird aktuell die Konformanz mit den Beschränkungen zumeist auf der Ebene des C Codes überprüft. Bei Nichteinhaltung der Regeln gibt es teilweise lange Rückkopplungschleifen, um die Ursachen für die Verstöße zu identifizieren und abzustellen. Daher besteht mittelfristig das Ziel, die Prüfungen weitestgehend auf Modellebene vorzunehmen. Aufgabe der Codegeneratoren wäre es dann zu gewährleisten, dass 'sichere' SL Modelle in 'sicheren' C Code zu übersetzt werden (Abb. 3).

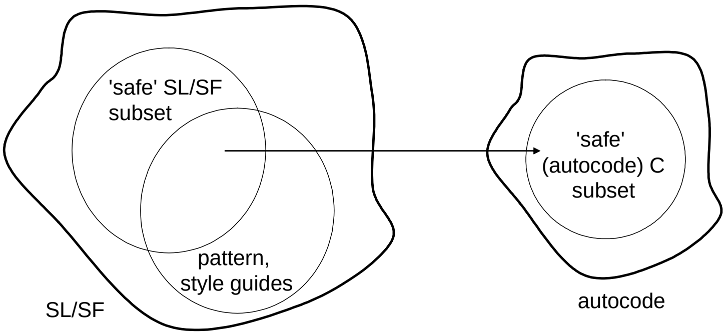


Abbildung 3: Zusammenhang zwischen 'sicheren' Sprachteilmengen auf Modell- und Codeebene

3.3 Safety Pattern

Bewährte Methoden und Maßnahmen, die im Rahmen der klassischen Programmierung zur Gewährleistung der Sicherheit eingesetzt werden, können auf die Modellebene übertragen und in Form so genannter Safety-Pattern für die Modellierung in standardisierter Form eingesetzt werden. Erfahrungswissen in Form erprobter Modellkonstrukte kann ebenfalls in Pattern-Form aufbereitet werden. Negativ-Pattern können ungeeignete Modellkonstrukte darstellen.

Auf diese Weise können in den einzelnen Organisationen vorhandene Richtliniensammlungen um spezielle *Safety Pattern* erweitert werden. Von zentraler Bedeutung hierbei ist, dass Nutzen und Anwendungsbereich fallweise erläutert werden.

Im Rahmen der Modell-basierten Entwicklung kann eine höhere Fehlertoleranz dadurch erreicht werden, dass der abzusichernde Teil des Nutzalgorithmus einmal mit Floatingpoint-Arithmetik und einmal in Fixpoint-Arithmetik ausgeführt wird. Bei geeigneter Konfiguration des Codegenerators können so unterschiedliche C-Code Fragmente erzeugt werden, die mit den gleichen Eingabewerten zum einen auf einer Floatingpoint-ALU und zum anderen auf einer Fixedpoint-ALU ausgeführt werden. Über einen geeigneten Komparator werden die Ergebnisse der beiden Berechnungen dann miteinander verglichen. Das in Abb. 4 dargestellte Safety-Pattern verdeutlicht das Prinzip.

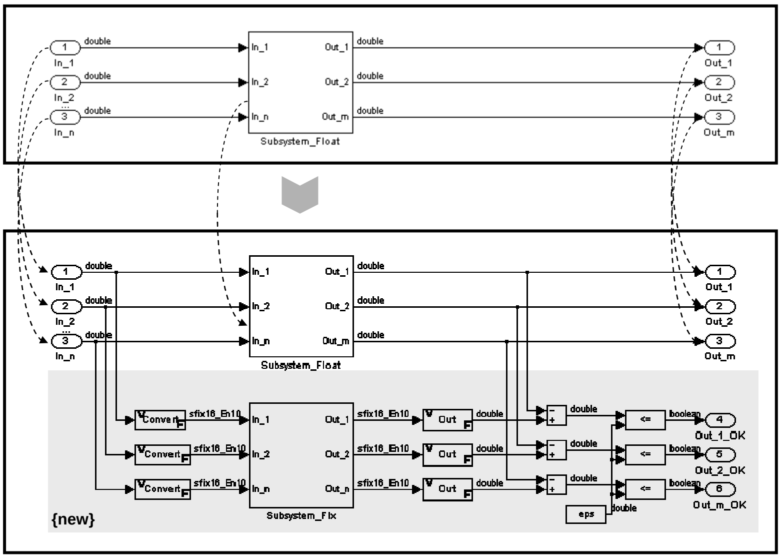


Abbildung 4: Safety-Pattern zur Erhöhung der Fehlertoleranz

Die Anwendung von Safety- und sonstigen Modellierungspattern führt zu einer weiteren Beschränkung der Modellierungssprache (vgl. Abb. 3)

Der Einsatz von Modellierungsrichtlinien und Safety Pattern sollte nach Möglichkeit werkzeugunterstützt werden. Notwendig sind zum einen eine Infrastruktur für anwenderfreundliche Aufbereitung solcher Guidelines mit Suchmöglichkeiten und zum anderen Guideline-Checker zur weitgehend automatischen Überprüfung der Guidelines.

Für den ersten Teil liegen positive Erfahrungen mit web-basierten Frontends für Verwaltung von und Zugriff auf Guidelinesammlungen vor. Exemplarisch sei hier die e-Guidelines[4] Infrastruktur [CFP05] genannt.

Die Einhaltung von Guidelines kann mit Werkzeugen wie MINT[5] oder dem Model-Advisor[6] teilautomatisiert werden. Wünschenswert wären eine Verknüpfung von Guidelinedatenbasen und entsprechenden Guideline-Checkern sowie abstrakte Sprachen zur Beschreibung der abzuprüfenden Regeln.

[4] www.e-guidelines.org
[5] www.ricardo.com
[6] www.mathworks.com

Forschungsgegenstand sind komplexere Modellanalysatoren und –transformatoren, die z.B. in Modellen automatisch Bad Smells oder mögliche Anwendungskontexte für Positiv-Pattern identifizieren und dann regelbasierte Modelltransformationen interaktiv ermöglichen. Auf diese Weise könnten auch moderne Programmieransätze wie die aspektorientierte Programmierung auf Modellebene verankert werden. Als Beispiel kann das oben aufgeführte Safety-Pattern dienen, bei dem die zusätzliche Funktionalität weitgehend additiv zu einem vorhandenen Modell hinzugefügt werden kann.

3.4 Nachweis der standardkonformen Entwicklung

Standards wie IEC61508 und ISO/WD 26262 schlagen für die einzelnen Sicherheits-Integritätslevel Techniken und Maßnahmen vor, die in den einzelnen Entwicklungsphasen zum Einsatz kommen sollen. Im Rahmen eines konkreten Projektes ist dabei zu dokumentieren, welche Maßnahmen wie umgesetzt wurden bzw. welche Ersatzmaßnahmen zur Anwendung kamen.

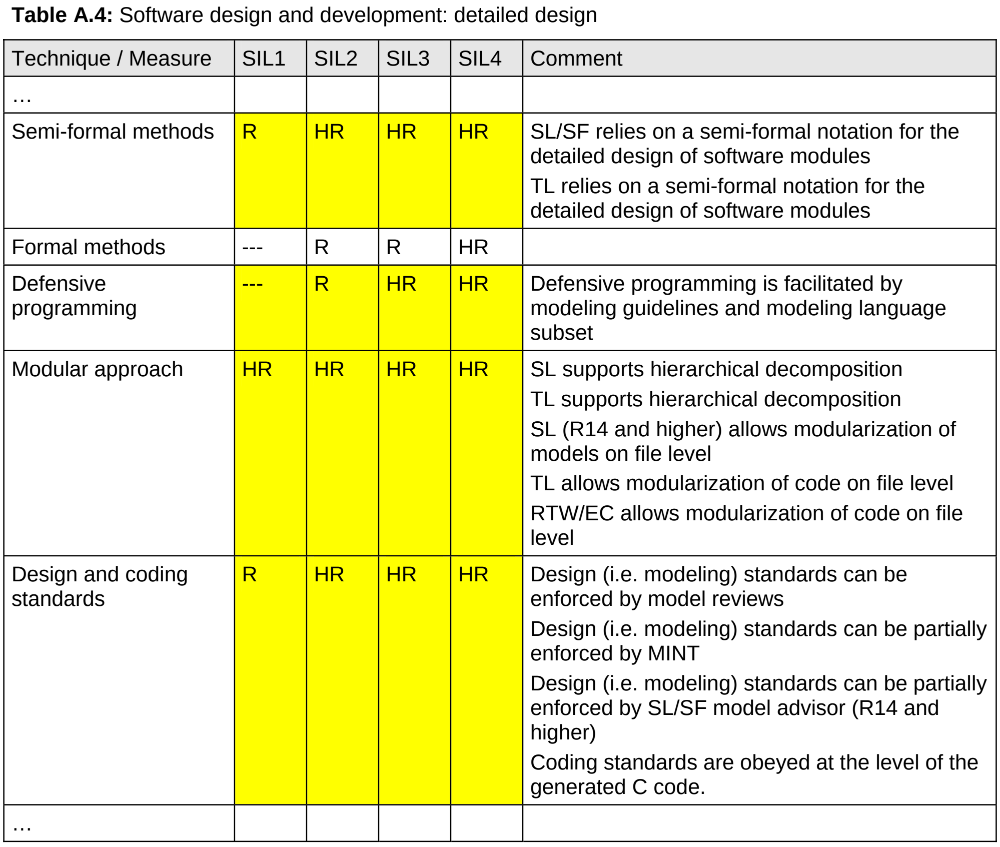


Table A.4: Software design and development: detailed design

Technique / Measure	SIL1	SIL2	SIL3	SIL4	Comment
…					
Semi-formal methods	R	HR	HR	HR	SL/SF relies on a semi-formal notation for the detailed design of software modules TL relies on a semi-formal notation for the detailed design of software modules
Formal methods	---	R	R	HR	
Defensive programming	---	R	HR	HR	Defensive programming is facilitated by modeling guidelines and modeling language subset
Modular approach	HR	HR	HR	HR	SL supports hierarchical decomposition TL supports hierarchical decomposition SL (R14 and higher) allows modularization of models on file level TL allows modularization of code on file level RTW/EC allows modularization of code on file level
Design and coding standards	R	HR	HR	HR	Design (i.e. modeling) standards can be enforced by model reviews Design (i.e. modeling) standards can be partially enforced by MINT Design (i.e. modeling) standards can be partially enforced by SL/SF model advisor (R14 and higher) Coding standards are obeyed at the level of the generated C code.
…					

Abbildung 5: Umsetzung der Methoden und Maßnahmen der IEC61508 im Rahmen der Modell-basierten Entwicklung

Der hierfür projektspezifisch anfallende Aufwand kann wirkungsvoll reduziert werden, wenn entsprechende generische Dokumente (Templates) für typische Modell-basierte Entwicklungsprojekte mit der in der Organisation verwendeten Toolkette bereitgestellt werden, die dann projektspezifisch instantiiert werden.

Abb. 5 zeigt einen exemplarischen Ausschnitt einer kommentierten IEC61508 Maßnahmen-Tabelle, die in einer separaten Spalte generische Anmerkungen zur Umsetzung im Rahmen Modell-basierter Entwicklungsprojekte enthält. Eine analoge Vorgehensweise ist auch im Rahmen des kommenden ISO/WD 26262 möglich.

4 Zusammenfassung

Der Beitrag hat skizziert, welche aufeinander abgestimmten Maßnahmen zum abgesicherten Einsatz modell-basierter Techniken für sicherheitsrelevante automobile Anwendungen erforderlich sind. Erschwert durch den Einsatz nicht formal spezifizierter Modellierungssprachen wie Simulink / Stateflow müssen eine Reihe von informellen Ansätzen, wie etwa die Definition einer sicheren Teilmenge der Modellierungssprache oder Safety Pattern, mit einander kombiniert werden. Die dazu erforderliche Werkzeugunterstützung liegt in Teilen vor. Insbesondere die Nutzung von Techniken der Modelltransformation wird den Grad der Absicherung der Einhaltung der Regeln jedoch noch deutlich erhöhen und die notwendigen Nachweise beschleunigen. Unter Einsatz der genannten Maßnahmen und Techniken kann dann die Konformität mit Sicherheitstandards belegt werden.

Literaturverzeichnis

[ASK04] A. Agrawal, G. Simon, G. Karsai: Semantic Translation of Simulink/Stateflow Models to Hybrid Automata Using Graph Transformations. ENTCS 109 (2004): 43-56

[CCM+03] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis: Translating Discrete-Time Simulink to Lustre. LNCS 2855 (2003): 84-99

[CFG+05] M. Conrad, I. Fey, M. Grochtmann, T. Klein: Modellbasierte Entwicklung eingebetteter Fahrzeugsoftware bei DaimlerChrysler. Informatik Forsch. Entw. (2005) 20:3-10

[CFP05] M. Conrad, I. Fey, H. Pohlheim: eGuidelines – A Tool for Managing Modeling Guidelines. International Automotive Conference, Detroit (US), Jun. 2005

[DC05] H. Dörr, M. Conrad : Modellbasierte Entwicklung sicherheitsrelevanter eingebetteter Systeme. Eingeladener Vortrag, MBEES'05, Dagstuhl, Jan. 2005

[MISRA04] MISRA-C:2004 Guidelines for the Use of the C Language in Critical Systems. Oct. 2004

[MSR02] MSR Working group MEGMA: Standardization of library blocks for graphical model exchange. 2001

[SSC+04] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, F. Maraninchi: Formal languages: Defining and translating a "safe" subset of simulink/stateflow into lustre. Proc. 4. ACM Int. Conf. on Embedded Software, Sept. 2004

Petri Net Model Synthesis from Scenarios

Jörg Desel

Lehrstuhl für Angewandte Informatik
Katholische Universität Eichstätt-Ingolstadt
85071 Eichstätt
joerg.desel@ku-eichstaett.de

Abstract: This paper argues that the development of embedded system can be based on model synthesis from a formal specification of desired scenarios. This claim is substantiated by experiences of the author with an automotive project. In this project, a specific class of Petri nets was applied, which is shortly introduced in the paper. After motivating that model synthesis in the Petri net setting is of practical importance, we sketch the state-of-the-art in this area and our current approach.

1 Introduction

Model based software engineering often starts its considerations with a given model which is assumed to specify the behavior of the system to be developed correctly and completely. This holds in particular for embedded systems, where the problem is originally formulated by engineers, whereas in information systems there exists much more experience in conceptual modeling on different levels, taking different abstraction levels, different aspects, different views, and in particular different modeling experience of the modelers into account. One of the main reasons for this difference is that the domain experts in embedded systems are typically engineers whereas the domain experts in information systems have various backgrounds, but in most cases are not very familiar with engineering or informatics. Therefore, model based software engineering for embedded system has to consider a completely different starting point than model based software engineering in other areas.

In a software engineering project with the automotive vendor Audi AG the given starting point was a formulation of scenarios on a quite detailed level. In other words, there was neither a formal nor an informal specification of the software to be constructed but rather sequences of steps to be performed in different cases. Very formal definitions of interfaces to other components were provided. Finally, some liveness and safety properties were given. These properties did obviously “hold for the given scenarios”.

So the aim in this kind of projects is *not* to construct algorithms from a given interface specification and given behavioral specifications but rather to synthesize an algorithm from the desired behavior (which had to be formalized first) and to prove the behavioral specification for this algorithm.

It is important to notice that the desired behavior, given by scenarios, is not formulated on the level of an algorithm but rather on the level of desired runs in a given environment of the algorithm to be constructed. Hence we have the following steps:

1. Construct a model of the environment or of the interface to the environment (in our case, this model is formulated in terms of a class of modular Petri nets).

2. Formalize the scenarios (in our case, a partial order of events is obtained from each scenario, partially equipped with timing information).

3. Synthesize a model of the algorithm from the formalized scenarios (in our case, this model is also formulated in terms of a class of modular Petri nets).

4. Formalize the desired properties of the algorithm (e.g. in temporal logics).

5. Prove that the composed models from Step 1 and Step 3 satisfy the properties formalized in Step 4.

6. Translate the model from Step 3 in software modules .

We concentrate in this contribution on Step 3, which turns out to be the most difficult one. In the automotive project mentioned before, the synthesis was done manually and the system model was validated by deriving the desired runs (and no undesired ones).

In the following section, we provide a sketch of the tasks in the above mentioned automotive project. The class of Petri nets applied is shortly introduced and some of the models from the project are shown.

The third section is devoted to the state-of-the-art in synthesis of Petri net models from given scenarios. It will turn out that this problem heavily depends on parameters such as knowledge of names of events, names of local states, number of scenarios, form of scenarios etc. Although these parameters are very important for the construction of algorithms, they seem to have no really relevant counterparts in the application world. On the other hand, problems like incompleteness or vagueness of the scenario specification, which turns out to be frequent phenomena in applications, cannot be handled by theory yet.

Finally, in the fourth section we discuss requirements for a more general and more applicable synthesis procedure. Moreover, we present our approach, which can be termed semi-automatic synthesis and which combines automatic synthesis and validation steps.

2 Models in an automotive project

This section gives a very rough sketch of a project we conducted together with engineers from the automobile vendor Audi AG in Ingolstadt. We also show some of the models that were developped in this project. These models are given as Petri nets with signal arcs, a language developed within the project SPECIMEN[1] (see [DHJ+04]). For details, see [DJL+03] and [DMN04].

The language we use is a class of *modular high-level Petri nets*, where the domain of (some) places is the set of real numbers. These places represent e.g. sensor values. They are represented by double circles. Modules are composed by means of two special arcs; *event arcs* and *read arcs*. An event arc connects two transitions and has the following semantics: the second transition fires if and only if it is enabled in its module and the first transition fires. In other words, either the first transition fires alone (if the second transition is not enabled), or both transitions fire together in one step. Event arcs are denoted by zigzag arcs. Arcs with two arrowheads represent read arcs. A transition with an attached read arc can only fire if the corresponding place carries a token. The marking of that place is not changed in case of low-level places. It can be overwritten in case of high-level places, where two variables at the arc denote the old and the new value, respectively. Finally, there are *time annotations* at some arcs which denote that the corresponding transition will fire after it was enabled for the time period indicated by the annotation.

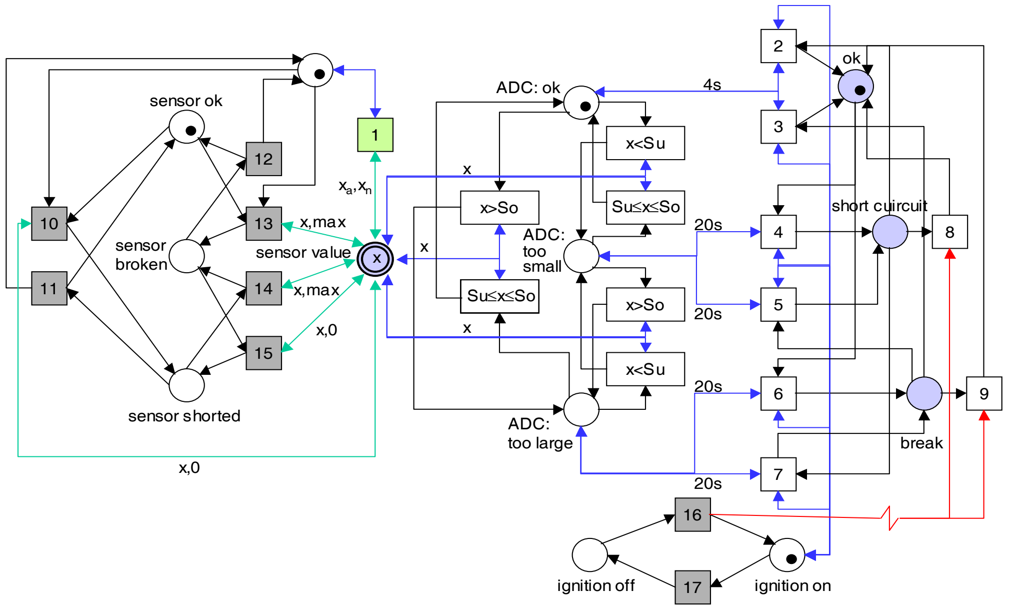


Figure 1: A Petri net model of an algorithm for sensor diagnosis

[1] SPECIMEN was supported by the German Research Foundation (DFG) as part of the Priority Program "Software Specification Techniques for Applications in Engineering"

Figure 1 shows a model of three components derived within the project. The left hand component models the physical sensor, which is assumed to be in exactly one of the three states: OK (it works correctly), broken (very high value), shorted (very low value). The six transitions 10 to 14 represent the possible state transitions. Transition 1 represents the fact that a sensor can change its sensor value as long as it is in the state OK. It is important to notice that this component is only related to other modules by means of the common place "sensor value". So the state of the sensor cannot be seen by the outside, but it can only be diagnosed by reading the sensor value.

The small model component at the bottom of the figure represents another physically existing entity: the ignition control which is governed by the driver. The diagnosis algorithm should only work when ignition is *on*. This state can be read by arbitrary events. Turning ignition *off* forces either the occurrence of Transition 8 or the occurrence of Transition 9.

The remaining part was to be constructed. It was specified by means of scenarios given in natural language. Their formalization in our language is shown in Figure 2. This figure shows three partially ordered runs of the Petri net of Figure 1. Our notion of a partially ordered run is a conservative generalization of the usual notion of runs (see e.g. [Rei98]). Notice that our runs have elements connected by a read arc. The partial order is obtained by the transitive closure of the relation given by regular arcs, extended by ordering events that read from a place *after* all events that precede that place and *before* all events that succeed the event. Instead of event arcs, a run can contain events that represent the occurrence of steps of transitions related by event arcs. Real values can be represented in runs by a variable, representing classes of values which lead to identical behaviour. Finally, time annotation is just transferred to the level of runs.

Actually, we did not obtain the runs presented here directly from the specifications. Instead, we constructed the Petri net modules of the physical existing parts first. Then we constructed those parts of the runs that correspond to the respective given scenarios. We further developed complete runs and the missing module (representing the algorithm) in parallel, ending up with a complete model and with runs that are runs of this model in a formal way. Finally, we proved further desired safety properties of the algorithm module using standard Petri net verification techniques.

The main experience from this project was that our approach was very well suited for specification purposes. The engineers from our project partner accepted the modeling language because they apparently had no problems in understanding the language and the models given in this language. The obtained models of algorithms were used as precise specifications of the algorithms.

The approach suffered from missing tools. Whereas drawing tools, verification tools and tools for generation of runs could quickly be added to our existing VIPtool environment [VIPtool], there exist no sufficient theory for synthesizing the final model from the runs. As shown in the following section, existing approaches depend on a precise knowledge about all elements of the (all) runs, whereas the natural language specification only gave rise to some elements of the runs and the ordering relation between them.

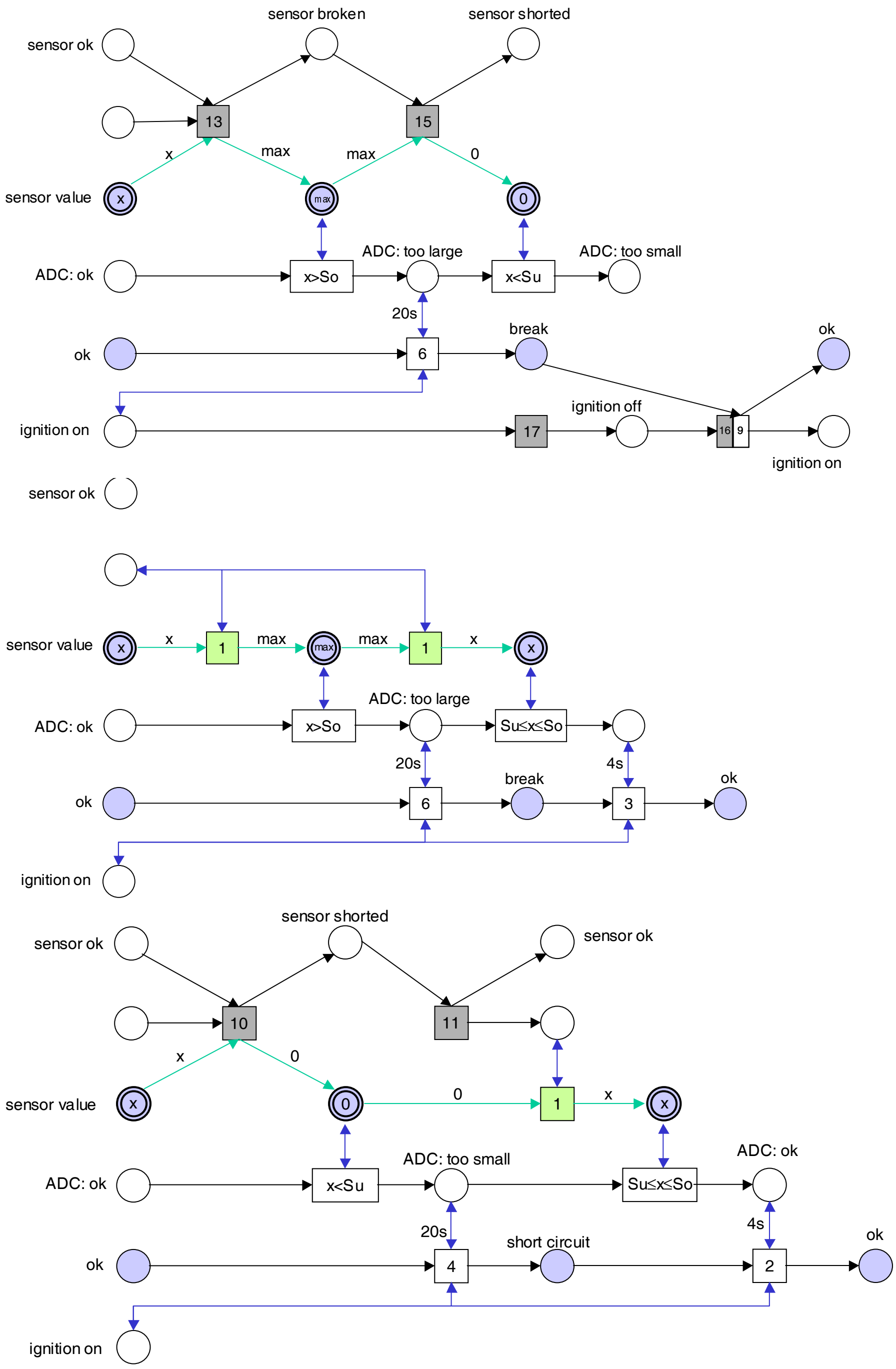


Figure 2: Specification by three scenarios

3 Synthesis of Petri net models from runs

The so-called synthesis problem of Petri nets is the problem of automatically deriving a Petri net model from its behavior. The most prominent approach starts with a transition system representing behavior and constructs a Petri net such that the state space of this Petri net is isomorphic to the transition system (see [ER89] and [DR96] for the synthesis of elementary net systems and [BD98] for more general cases). Isomorphism means that event labels are respected; so the name of the events must be known whereas the states are anonymous.

Synthesis from sequential runs can be reduced to synthesis from transition systems as long as the state space can be constructed from sequential runs. Since runs are usually given by occurrence sequences enabled by the initial marking, the crucial point in the translation is to identify those pairs of occurrence sequences that lead to the same marking (without knowing this marking). If all possible sequences are given, results from automata theory can be used to solve this problem.

In the approach presented in this paper, the Petri net is constructed not from sequential runs but rather from partially ordered runs, given by extended occurrence nets. The partial order represents the causality relation between events and conditions. The first publication on this topic is [Smi91]. Here all runs are assumed to be given. In the area of workflow nets, [DE00] and [DDA05] start with some runs that the system should have, whereas it might have additional ones. The core idea of all these approaches is given by foldings of occurrence nets representing runs, i.e., by construction of suitable classes of nodes of occurrence nets which then represent the nodes of the system net.

4 Combining automatic synthesis and manual construction

Synthesizing Petri nets from their runs is still a field where a lot of work has to be done. However, all existing approaches assume a formal representation of all or of some runs, where each run is given by its sequence of events or by a partial order of its events. Some approaches assume moreover that the places of the system net are represented as conditions in the occurrence net (sometimes it is assumed that these conditions can be identified, sometimes only knowledge about their number is known).

Experiences with the application of these approaches show that the above assumptions are too optimistic. Instead, only some events of the runs are given, only some conditions are known, and the order between events and conditions is only partially available.

A typical example for missing conditions is that necessary routing elements are often not included in the specification of scenarios. A scenario description might contain three events that occur sequentially, hence they are causally ordered. An obvious way to model this situation is to include two conditions between these events. Another scenario might only contain the first and the third event whereas the second event does not occur in this scenario. Then a "dummy event" has to be added which moves a token from the post-condition of the first event to the pre-condition of the third event. There is no way to guess this condition when formalizing the second scenario. On the other hand, all existing formal approaches to Petri net synthesis cannot handle the scenarios without the dummy event because the pre- and post-sets of the elements do not match. Another example is given by scenarios with events that can occur either sequentially or concurrently.

For the above reason, our current research goes in two directions: first we try to develop synthesis algorithms that can cope with the above mentioned situations. Instead of a single partial order, scenarios are modeled by sets of events with two partial orders; the minimal causality (ordered pairs of events must happen sequentially) and the maximal causality (unordered pairs of events must occur concurrently). Each run of the system to be synthesized that includes these events with a causality relation in between the two specified relation satisfies this scenario specification. Moreover, formalized scenarios are refined in such a way that they match each other, i.e., such that pre- and post-conditions of events agree in all runs. To this end, additional net elements have to be added.

The other research direction, which is more application-oriented, assumes that only a part of the system can actually be synthesized. Modifying and completing alternately the system and its runs will finally lead to the desired system. During this approach, erroneous scenarios are identified and so the initial formalization step is supported. Moreover, when starting with some intended runs, usually some of the runs not explicitly specified should be allowed as well, whereas other runs represent forbidden behavior. Our approach allows to interactively adjust the system accordingly. For this research direction, we are currently extending the functionality of our tool Viptool.

References

[BD98] Badouel, E.; Darondeau, P.: Theory of regions. In: W. Reisig, G. Rozenberg (Eds.): Lecture Notes on Petri Nets, Part 1 (Basic Models), LNCS 1491, Springer-Verlag Berlin Heidelberg 1998, pp.529-586

[DDA05] van Dongen, B.F.; Desel, J.; van der Aalst, W.M.P.: Aggregating causal runs to workflow nets. Submitted paper (2005)

[DE00] Desel, J.; Erwin, T.: Hybrid workflows: looking at workflows from a run-time perspective. Computer Systems Science and Engineering 5:291-302, 2000

[DJL+03] Desel, J.; Juhás, G.; Lorenz, R.; Milijic, V.; Neumair, C.; Schieber, R.: Modellierung von Steuerungssystemen mit Signal-Petrinetzen*:* Eine Fallstudie aus der Automobilindustrie. In: E. Schnieder (Hrsg.): Entwurf komplexer Automatisierungssysteme (EKA 2003), Universität Braunschweig, pp.273-295 (2003)

[DHJ+04] Desel, J.; Hanisch, H.-M.; Juhás, G.; Lorenz, R.; Neumair, C.: A guide to modelling and control with modules of signal nets. In: H. Ehrig et. al.: Integration of Software Specification Techniques for Applications in Engineering. LNCS 3147, Springer-Verlag Berlin Heidelberg 2004, pp.270-300 (2004)

[DMN04] Desel, J.; Milijic, M; Neumair, C: Model validation in controller design. In: J. Desel, W. Reisig, G. Rozenberg (Eds.): Lectures on Concurrency and Petri Nets. LNCS 3098, Springer-Verlag Berlin Heidelberg 2004, pp.467 – 495

[DR96] Desel, J.; Reisig, W.: The synthesis problem of Petri nets. Acta Informatica 33:297-315 (1996)

[ER89] Ehrenfeucht, A.; Rozenberg, G.: Partial (Set) 2.Structures – Part 1 and Part 2. Acta Informatica 27(4):315-368 (1989)

[Rei98] Reisig, W.: Elements of Distributed Algorithms. Springer-Verlag (1998)

[Smi91] Smith, E.: On net systems generated by process foldings. In: G. Rozenberg (Ed.): Advances in Petri Nets, LNCS 524, Springer-Verlag Berlin Heidelberg 1991, pp.253-246

[VIPtool] http://www.informatik.ku-eichstaett.de/projekte/vip/

Abdeckungskriterien in der modellbasierten Testfallgenerierung: Stand der Technik und Perspektiven

Mario Friske, Bernd-Holger Schlingloff
Fraunhofer FIRST
Kekuléstraße 7
D-12489 Berlin
{mario.friske,holger.schlingloff}@first.fhg.de

Abstract: Zur Beurteilung der Qualität von Testsuiten ist der Abdeckungsgrad ein wichtiges Kriterium. Beim modellbasierten Blackbox-Test wird dabei nicht die Abdeckung im Ziel-Code, sondern in der Spezifikation gemessen. Wir beschreiben anhand eines konkreten Beispiels die von aktuellen Testfallgenerierungswerkzeugen erreichte Abdeckung und argumentieren, dass eine zusätzliche Metastrategie für die Kombination von Abdeckungskriterien erforderlich ist.

1 Einleitung

Die modellbasierte Entwicklung stellt ein neues Paradigma in der Entwicklung eingebetteter Systeme dar. Ausgehend von einer Spezifikation der Anforderungen in natürlicher Sprache wird dabei ein plattformunabhängiges Funktionsmodell erstellt, welches schrittweise zu einem Implementierungsmodell erweitert wird. Aus diesem Implementierungsmodell kann dann der ausführbare Code für die eingebettete Zielplattform automatisch generiert werden. Als Modellierungsformalismen werden dabei häufig Zustandsautomaten der UML oder Signalflussgraphen aus Matlab/Simulink® verwendet.

Zunehmend wird das modellbasierte Paradigma auch verwendet, um Testsuiten automatisch zu generieren: aus dem Funktionsmodell werden in diesem Fall direkt Testfälle erzeugt, mit denen die weiteren Verfeinerungen und Implementierungen getestet werden. Wenn diese Vorgehensweise begleitend zur Systementwicklung eingesetzt wird, können die Tests dazu dienen, die Korrektheit der einzelnen Verfeinerungsschritte zu überprüfen.

Aus verschiedenen Gründen ist die Implementierung jedoch oft nur als Blackbox verfügbar. In diesem Fall dient das Funktionsmodell als Referenz, gegenüber der die Korrektheit von Anforderungen getestet wird. Das Funktionsmodell ist dabei im Allgemeinen eher als deklarative denn als imperative Spezifikation formuliert, in der das „was“ und nicht das „wie“ eines Systems beschrieben wird. Beispielsweise werden unabhängige Aktivitäten wie das Drücken einer Taste durch den Benutzer und das Einschalten eines Motors durch das System in der deklarativen Spezifikation als parallele Aktionen spezifiziert, die kausal voneinander abhängen. Funktionsmodelle für den modellbasierten Test beschreiben also im Allgemeinen eher die Anforderungen an ein System als die Art und Weise, in der die Tests durchgeführt werden sollen.

Es erhebt sich die Frage nach der Vollständigkeit der automatisch generierten Testsuiten. Die im Bereich der Luft- und Raumfahrt häufig verwendete Norm DO-178 B erfordert für Systeme der höchsten Kritikalitätsstufe eine vollständige Testabdeckung nach den Kriterien *„Modified Condition Decision Coverage (MC/DC)"*, *„Branch/Decision Coverage"* und *„Statement Coverage"*. Im Automobilbereich wird häufig auf die Norm IEC 61508 zurückgegriffen, während im Bahnbereich die CENELEC-Norm EN50128 maßgeblich ist. Beide Normen verlangen für Systeme der Sicherheitsstufe SIL4, dass Funktions- bzw. Blackbox-Tests durchgeführt werden, wobei eine Grenzwertanalyse und Äquivalenzklassenbildung dringend empfohlen wird. Aus Sicht eines Testingenieurs ist es wünschenswert, dass die generierten Testsuiten eine definierte Überdeckung der Anforderungen garantieren können.

In [GS05] wird ein Überblick über Abdeckungskriterien für den modellbasierten Test gegeben und die abdeckungsorientierte Vorgehensweise mit statistischen Testmethoden verglichen. Im Folgenden werden Abdeckungskriterien an einem konkreten Beispiel evaluiert und Erweiterungsmöglichkeiten aufgezeigt.

2 Stand der Technik

Wir wollen den aktuellen Stand der Technik in der modellbasierten Testfallgenerierung an dem in Abbildung 1 dargestellten Beispiel erläutern. Es handelt sich um eine auf eine Bewegungsachse beschränkte Realisierung der in [HP02] spezifizierten Sitzsteuerung. Die Bewegung des Sitzes in einer Achse soll über einen Sitztaster mit drei Tasterstellungen gesteuert werden. Wird der Taster betätigt, soll der Sitz in die entsprechende Richtung bewegt werden. Bei Erreichen der jeweiligen Endposition soll die Bewegung unterbrochen werden. Eine weitere Bewegung in diese Richtung soll erst wieder möglich sein, nachdem der Sitz per Taster in die entgegengesetzte Richtung zurückgefahren wurde.

Die Modellierung des Statecharts erfolgte in Anlehnung an die in [DKvK$^+$02] beschriebenen Richtlinien. Das Statechart hat einen orthogonalen Zustand, welcher vier nebenläufige Regionen hat. In diesen sind jeweils die Sensoren *Taster* und *Endposition*, die eigentliche *Steuerung* und der Aktuator *Motor* modelliert. Wird eine durch die Sensoren überwachte Größe geändert, so erfolgt der entsprechende Zustandswechsel. Die Änderung wird der *Steuerung* durch ein zugehöriges Ereignis signalisiert. Die *Steuerung* kontrolliert den Zustand des Aktuators *Motor* ebenfalls durch Ereignisse.

In dem Modell ist auch eine Abhängigkeit modelliert, welche normalerweise im zugehörigen Umgebungsmodell realisiert werden würde, hier jedoch in direkt integriert wurde: Eine Bewegung des Sitzes in der Endposition in die Gegenrichtung hat einen Übergang von der Endposition in die Normalposition zur Folge.

Aus diesem Statechart sollen nun mit einem Testfallgenerator Testfälle erzeugt werden. Ein Testfall ist dabei eine Sequenz von Ereignissen, welche sowohl Eingaben an das zu testende System als auch dessen Reaktionen enthält. Zur automatischen Erstellung solcher Testfälle verwenden wir im Folgenden das Werkzeug ATG [IL], welches als Zusatz zur modellbasierten Entwicklungsumgebung Rhapsody® von I-Logix™ erhältlich ist. Andere

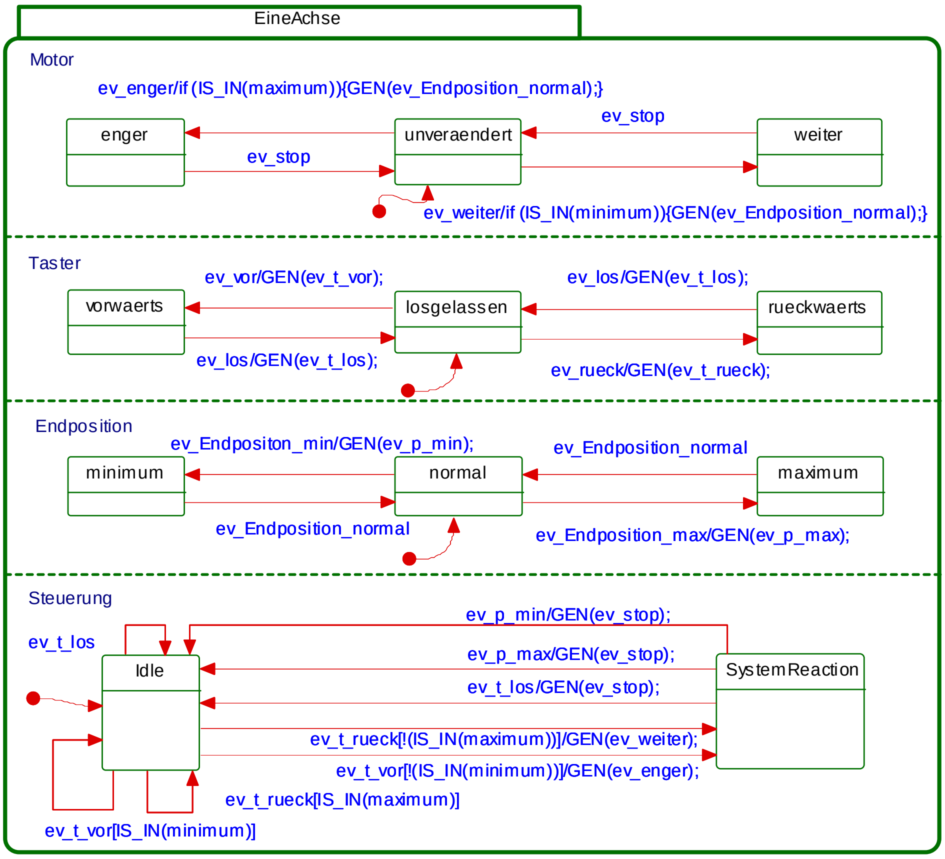


Abbildung 1: Statechart einer vereinfachten Sitzsteuerung

Werkzeuge mit ähnlicher Funktionalität sind z.B. Conformiq® [Con] und Reactis® [Rea]. ATG analysiert das UML-Modell und den daraus generierten C++ Code und generiert einen Satz von Testfällen. Die erreichbare Abdeckung ist MC/DC auf dem generierten Code. Dieses Abdeckungskriterium auf Quelltextebene umfasst die Modellebenenkriterien Transitions-, Ereignis- und Zustandsüberdeckung.

Im ATG werden zunächst die an den Schnittstellen zu betrachtenden Ereignisse festgelegt. Im Fall der Sitzsteuerung sind das nur die Ereignisse, die eine Zustandsänderung der Sensoren und Aktuatoren bewirken. Daraus erzeugt ATG Testziele auf Modell- und Code-Ebene. Auf Modellebenen wird für jedes der Modellelemente *Zustand*, *Transition*, *Operation* und *Ereignis* ein Testziel definiert. Auf Code-Ebene wird für jede Wertekombination in Bedingungen, die zum Erreichen von MC/DC erforderlich ist, ein Testziel erstellt.

Im Beispiel ergeben sich so 96 Testziele. Davon werden beim automatischen Generieren der Testfälle 81 erreicht. Im Detail sind das 17/17 Zustände, 24/24 Transitionen, 0/0 Operationen, 14/14 Ereignisse und 26/41 Code-Abdeckung. Daraus resultiert eine An-

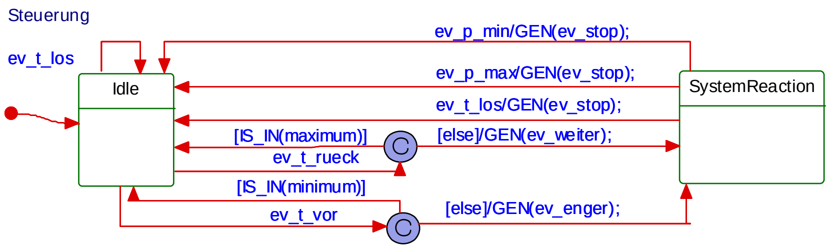


Abbildung 2: Modifikation des Statecharts zur Erhöhung der Testabdeckung

weisungsüberdeckung von 180/213. Bei der Analyse der Testfälle zeigt sich, dass einige Testziele nicht erreicht werden können, da an bestimmten Stellen im Quelltextes innerhalb verschachtelter Fallunterscheidungen sowohl Bedingungen als auch deren Negation generiert werden:

```
//## transition 17
if(!(IS_IN(maximum)))
    { ... }
else
    {
        //## transition 23
        if(IS_IN(maximum))
            { ... }
    }
```

Die Bedingung in der zweiten if-Anweisung kann nicht negiert werden. Daher ist das Kriterium MC/DC für diese Anweisung nicht erfüllbar.

Durch die in Abbildung 2 dargestellte alternative Modellierung unter Verwendung von Konditionskonnektoren lässt sich dieses Problem vermeiden. Die bei der Testfallgenerierung erreichte Abdeckung erhöht sich auf 83/96 Testziele, d.h. die Code-Abdeckung erhöht sich auf 26/39 oder 66 Prozent. Zustände, Transitionen, Operationen und Ereignisse werden nach wie vor vollständig abgedeckt. Im Falle der Transitionen sind es nun zwei mehr, d.h. 26/26. Da aus einem geänderten Modell auch anderer Code generiert wird, schlägt sich die Verbesserung nicht notwendigerweise in der gemessenen Anweisungsüberdeckung nieder. Im Beispiel ergibt sich sogar ein leicht geringerer Wert von 178/211.

Die 13 nicht erreichten Testziele bezüglich Code-Abdeckung erklären sich folgendermaßen: viermal wird das Code-Abdeckungs-Testziel „!IS_IN(EineAchse)“ nur teilweise erreicht, da der Zustand *Eine_Achse* nicht verlassen wird. Fünfmal wird für die den einzelnen Zuständen zugeordnete Funktion *dispatchEvent* keine vollständige *Switch Coverage* erreicht. Viermal wird kann der Funktionsaufruf „EineAchse_Exit“ nicht erreicht werden, da der Zustand *EineAchse* nicht verlassen wird. Diese 13 nicht erreichten Testziele resultieren aus der Art, wie Rhapsody den Code aus dem Statechart generiert und sind auch nicht ohne weiteres zu erreichen. Auf dem erreichbaren Code kann folglich mit den generierten Testfällen 100% Zustandsüberdeckung, 100% Transitionsüberdeckung, 100% Ereignisüberdeckung und 100% MC/DC-Abdeckung erzielt werden.

Beim Export wird pro Testziel der resultierende Testfall in einer entsprechend benannten Datei abgelegt. Dadurch entstehen 92 Testfälle. Da verschiedene Testziele zu gleichen Ereignissequenzen führen können, enthalten die Testfälle Duplikate und Inklusionen. Das Format der generierten Testfälle unterscheidet Ein- und Ausgaben, beinhaltet Zeitstempel und ist in der Lage, mit mehr als einem zur Verarbeitung anstehenden Ereignis umzugehen. Da im Beispiel nur die Abfolge der Ereignisse von Interesse ist, lassen sich die Testfälle vereinfacht als Ereignissequenzen darstellen. Nachdem Duplikate und Inklusionen durch einen Postprozessor entfernt wurden, bleiben die in Abbildung 3 gezeigten 13 Testfälle übrig.

```
TC1:
ev_Endpos_max
ev_vor
ev_enger
ev_Endpos_norm

TC2:
ev_Endpos_min
ev_rueck
ev_weiter
ev_Endpos_norm

TC3:
ev_Endpos_norm

TC4:
ev_los

TC5:
ev_rueck
ev_weiter
ev_Endpos_max
ev_stop

TC6:
ev_rueck
ev_weiter
ev_los
ev_stop

TC7:
ev_vor
ev_enger
ev_Endpos_max
ev_stop

TC8:
ev_Endpos_min
ev_Endpos_norm

TC9:
ev_Endpos_max
ev_Endpos_norm

TC10:
ev_vor
ev_enger
ev_los
ev_stop

TC11:
ev_rueck
ev_weiter
ev_Endpos_min
ev_stop

TC12:
ev_Endpos_max
ev_rueck
ev_los

TC13:
ev_Endpos_min
ev_vor
```

Abbildung 3: Aus dem Statechart gemäß Abdeckungskriterium MC/DC generierte Testfälle

3 Perspektiven

Obwohl die so gewonnene Testsuite eine 100%ige Testabdeckung nach MC/DC-Kriterium auf dem Modell erreicht, ist sie intuitiv nicht ausreichend. Etliche Eigenschaften, die bei einer manuell erstellten Testsuite geprüft würden, sind in dieser automatisch erstellten Testsuite nicht enthalten. Ein typisches darüber hinaus gehendes Kriterium ist beispielsweise die in den Anforderungen spezifizierte Eigenschaft, dass sich der Sitz nach Erreichen der Endposition auch bei erneuter Betätigung des Tasters nicht bewegt. Auf Grund der automatischen Generierung ist es nicht möglich, bestimmte als gefährlich vermutete Teilpfade im Statechart als Testfälle einzustellen. Ein anderer zu testender Aspekt ist beispielsweise: „Die Abschaltung in eine Richtung funktioniert auch noch beim zweiten Mal.“ Auch solche wiederholten Abläufe werden durch die in ATG implementierte Testerzeugungsstrategie nicht erfasst.

MC/DC auf dem Code wird in der Industrie oft als Abdeckungskriterium verwendet. Im Beispiel werden die Testfälle unter Verwendung des Kriteriums MC/DC aus dem Modell generiert. Wird das Modell als Spezifikation zum Black-Box-Test einer unbekannten Implementierung verwendet, so ist dies natürlich keine Aussage über die Codeabdeckung der Implementierung. Nach [Pos96] lassen sich Defekte in der Implementierung in die drei

Klassen *wrong code*, *missing code* und *extra code* einteilen. Die Bestimmung von *extra code* erfordert beim spezifikationsbasiertes Testen zusätzliche Codeabdeckungsmessungen. Das spezifikationsbasierte Testen ist besonders geeignet, *missing code* aufzudecken.

Für die Aufdeckung von Defekten der dritten Kategorie *wrong code* ist es ratsam, stärkere Kriterien als das von den aktuellen Testfallgeneratoren umgesetzte MC/DC zu verwenden. In [Bin99] werden solche Kriterien zum Black-Box-Test von Zustandsmaschinen diskutiert, u.a. *All n-Transition Sequences*, *All Round-trip Path* und *M-Length Signature*. Mit zunehmender Mächtigkeit der Strategien steigt die Zahl der abgedeckten Fehlerklassen. Zu jeder dieser Strategien lassen sich jedoch jeweils Fehlerklassen konstruieren, welche durch diese nicht abgedeckt werden können. Außerdem steigt die Anzahl der notwendigen Testfälle explosionsartig an. In [Bin99] wird der Versuch unternommen, die einzelnen Testverfahren nach den von ihnen abgedeckten Fehlerklassen zu klassifizieren. Eine solche abdeckungsorientierte Klassifikation entspricht wesentlich mehr der intuitiven Herangehensweise bei der manuellen Testfallerstellung als die üblichen verfahrensorientierten Klassifikationen von Testverfahren [Lig02].

In der Praxis werden häufig die angewandten Abdeckungskriterien durch die zur Verfügung stehenden Werkzeuge bestimmt. Aktuell verfügbare Werkzeuge unterstützen Kombinationen von Abdeckungskriterien und zugehörigen Generierungsstrategien nur unzureichend. In unserem Beispiel könnte eine kombinierte Strategie beispielsweise diese Kriterien umfassen: (a) MC/DC auf dem generiertem Code, (b) Erzeugen bestimmter Ereigniskombinationen und (c) Erzeugen von Sequenzen bestimmter Länge. Sofern die Ereignisse Parameter enthalten, ist die Kombinationen von Parametern ein mögliches zusätzliches Kriterium.

In einem Industrieprojekt haben wir eine solche kombinierte Strategie realisiert. Dabei werden zusätzliche Modellelemente in das Modell eingefügt, welche den Generator dazu bringen, weitere Testfälle zu erzeugen. Die so erhaltene Testsuite wird mit einem von uns entwickelten Postprozessor weiterverarbeitet, um zusätzliche Abdeckungskriterien zu erreichen.

Wir sind gerade dabei, die in diesem Projekt gewonnenen Erfahrungen zu verallgemeinern. Bei der manuellen Erstellung von Testsuiten geht ein Testingenieur oftmals von gewissen Risiken und Szenarien aus, die getestet werden sollen. Auch bei der automatischen Erzeugung von Testfällen aus Spezifikationen wird daher eine Metastrategie benötigt, um die Generierung zu steuern. Kriterien für die Testabdeckung sollten idealerweise auf der gleichen Abstraktionsebene wie in der modellbasierten Entwicklung spezifiziert werden können, d.h. das „was“ und nicht das „wie“ sollte im Vordergrund stehen. Eine Realisierungsmöglichkeit für die Angabe solcher Metastrategien besteht darin, die zu erzielenden Abdeckungskriterien, abzudeckenden Risiken und Kombinationen davon als *deklarative Testgenerierungsdirektiven* zu formulieren. Ein Beispiel für solche Direktiven sind die Kombinationsregeln im CTE/XL [LW00].

Aus dieser Idee ergeben sich eine Vielzahl von Fragestellungen, insbesondere im Hinblick auf Automatisierungsmöglichkeiten und Werkzeugunterstützung. Wie spezifiziert man zu testende Aspekte und zugehörige Abdeckungskriterien? In wie weit kann dies von Werkzeugen unterstützt werden? Wie ist eine Integration in bestehende Werkzeuge realisierbar?

Denkbar sind beispielsweise Werkzeuge, die Modell und Anforderungen analysieren und dem Tester modellspezifische Abdeckungskriterien zur Auswahl anbieten.

4 Zusammenfassung

In diesem Papier haben wir an Hand eines konkreten Beispiels die Verwendung des Werkzeuges ATG zur automatischen Generierung von Testfällen aus UML Zustandsautomaten beschrieben. Es zeigt sich, dass die resultierende Testsuite für modellbasierte Blackbox-Tests keinen intuitiv zufriedenstellenden Abdeckungsgrad bietet. Wir haben skizziert, wie automatisch generierte Testfälle durch zusätzliche Nachbearbeitung kombiniert werden können, um die Abdeckung zu verbessern. Allgemein sind nach unserer Auffassung beim spezifikationsbasierten Test risiko- und szenariogesteuerte Direktiven als Metastrategie für die automatische Testfallerzeugung wünschenswert.

Literatur

[Bin99] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Object Technology Series. Addison Wesley, 1999.

[Con] Conformiq Software Ltd. Conformiq Test Generator. http://www.conformiq.com/.

[DKvK+02] Christian Denger, Daniel Kerkow, Antje von Knethen, Maricel Medina Mora und Barbara Paech. Richtlinien - Von Use Cases zu Statecharts in 7 Schritten. IESE-Report Nr. 086.02/D, Fraunhofer IESE, 2002.

[GS05] Christophe Gaston und Dirk Seifert. Evalutating Coverage Based Testing. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker und Alexander Pretschner, Hrsg., *Model-Based Testing of Reactive Systems*, Jgg. 3472 of *Lecture Notes in Computer Science*, Seiten 293–322. Springer Verlag Berlin Heidelberg, 2005.

[HP02] Frank Houdek und Barbara Paech. Das Türsteuergerät - eine Beispielspezifikation. IESE-Report Nr. 002.02/D, Fraunhofer IESE, 2002.

[IL] I-Logix. Rhapsody Automatic Test Generator (ATG). http://www.ilogix.com/.

[Lig02] Peter Liggesmeyer. *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Spektrum Akadamischer Verlag, 2002.

[LW00] Eckard Lehmann und Joachim Wegener. Test Case Design by Means of the CTE XL. Proceedings of the 8th European International Conference on Software Testing, Analysis & Review (EuroSTAR 2000), Kopenhagen, Denmark, December 2000.

[Pos96] R. M. Poston. *Automating Specification-Based Software Testing*. IEEE Computer Society, Los Alamitos, 1st. Auflage, 1996.

[Rea] Reactive Systems Inc. Reactis. http://www.reactive-systems.com/.

Scenario-Based Verification of Automotive Software Systems

Matthias Gehrke, Petra Nawratil
Software Quality Lab (s-lab),
University of Paderborn,
D - 33095 Paderborn, Germany
[mgehrke|penaw]@uni-paderborn.de

Oliver Niggemann
dSPACE GmbH,
Technologiepark 25,
D - 33100 Paderborn, Germany
ONiggemann@dspace.de

Wilhelm Schäfer, Martin Hirsch*
Software Engineering Group,
University of Paderborn,
D - 33095 Paderborn, Germany
[wilhelm|mahirsch]@uni-paderborn.de

1 Introduction

Within the automotive industry, software engineering becomes more and more important. Especially component-based design is a popular approach in order to specify large, complex and reusable software systems.

The reuse of software components has become a major challenge for the automotive software development. By reusing components, manufacturers and suppliers can (*i*) revert to already tested software modules, thus minimizing potential software hazards, (*ii*) save development efforts and (*iii*) transfer product line approaches more easily into the world of software development. Several approaches exist to specify the structure, including interface definitions, for such reusable software components. AUTOSAR is the most well-known approach in the field of automotive software development.

In order to specify automotive systems with the component-based approach it is necessary to specify some aspects of the behavior of the components as well. More precisely, the interface descriptions of the components have to be enhanced with additional information, mainly timing aspects[1]. Just when the interfaces fit together, a feasible component structure arises.

After connecting the components, a large component structure arises and therefore a very complex structure of connected behavior models emerges. This complexity leads to several problems, because a wrong definition of the behavior models may invoke a not wanted behavior. At a specific degree of complexity, an engineer is not able to control the connected components and guarantee that no unwanted behavior will happen. So, some

*Supported by the University of Paderborn

[1] The internal structure of a software component will not be available, because that is the specific knowledge of the manufacturer and therefore constitutes one of his assets.

kind of automatic tests is needed, e.g. model checking.

This paper addresses one problem occurring when such reusable software components are used: How can be checked whether a system constructed from black boxes whose internals are unknown meets important constraints? This holds especially true for temporal requirements.

One solution is of course to put the software components on an evaluation board or electronic control unit (ECU) and have extensive runtime tests. This solution costs time and therefore money and still fails to verify all possible scenarios. It also assumes that component implementations already exist. But in many cases software developers first specify component interfaces which are then implemented by function developers. But even in this case the software developer may want to verify whether the designed system works correctly – presuming that the function developers fulfill all requirements.

A different approach solves these problems: component interface descriptions are enhanced with additional information, especially with timing models. Based on these information, formal verification methods such as model checking[2] are used to answer typical questions such as: (*i*) Given specific input, does the system reach a specific state within a given deadline? (*ii*) Does the system start up within a given time period? (*iii*) Do deadlocks exist?

These techniques of course do not replace runtime test; they only allow the early verification of some system properties. With this approach it is possible to check the behavior of an automotive software system in an early phase of the development process.

2 Example: Adaptive Cruise Control

The following example was developed within a project between the dSPACE GmbH and the Software Quality Lab (s-lab) of the University of Paderborn.

2.1 Structure

Figure 1 shows the simplified structure of an adaptive cruise control[3]. An adaptive cruise control keeps the car at a given constant speed. If an obstacle like another car makes the retention of the predefined velocity impossible or dangerous, the car's speed is reduced accordingly. The system accelerates the car automatically, if the obstacle disappears. Details may be found in [Rob03]. The system is specified with UML 2.0 [OMG03].

To simplify matters, not all software components and not all connections are shown. Nevertheless this example is typical for several reasons:

- The adaptive cruise control uses several electronic control units (ECUs). The reader may note that these ECUs are normally produced by different suppliers. In Figure 1 an adaptive cruise control unit (ACC ECU), an ESP (electronic stability control)

[2] In this paper we use the model checker Uppaal to test our approach.

[3] For several reasons, this example is far from being a realistic adaptive cruise control system: (*i*) intellectual property had to be protected, (*ii*) the example's purpose is comprehensibility, not completeness and correctness, and (*iii*) the authors do not develop adaptive cruise controls.

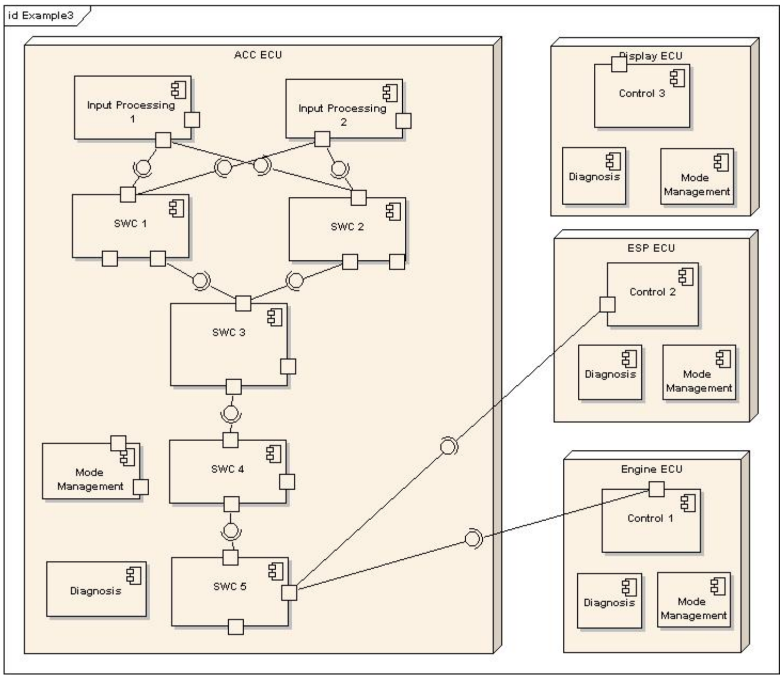


Figure 1: *A simplified Adaptive Cruise Control*

ECU[4], an engine ECU, and the display ECU are shown.

- The adaptive cruise control is a real-time, safety critical system. Its reactions have to meet strict timing constraints. Since any non-conformance to timing or behavioral requirements can result in fatal consequences, the highest safety standards are mandatory.

- The software is structured into (hierarchical) software components. This software structure defines the system's functionality while the distribution of software components to ECUs is responsible for the system's timing.

- The system's functionality is specified in more detail in form of timing models associated with the component interfaces (cf. Section 2.2).

2.2 Timing behavior

The next step is to specify the behavior of the interfaces of the components considering timing requirements. In this approach we use Timed Automata for specifying the timing description of the interfaces.

[4] responsible for the deceleration

The example in Figure 2 models the behavior of a sensor data processing component (*Input Processing 1* in Figure 1), i.e. a software module that gets new sensor data from some sensor hardware and sends it to other adaptive cruise control software modules (e.g. *SWC 1 and SWC 2* in Figure 1). Three states are modeled: (*i*) running, (*ii*) shutting_down, and (*iii*) not_running.

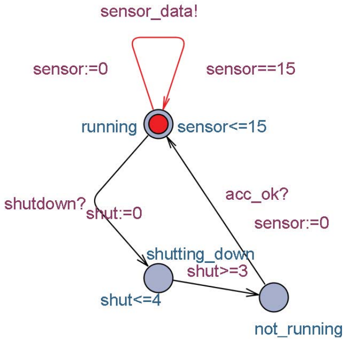


Figure 2: *A Timed Automaton in Uppaal*

The component sends new data every 15 time units (e.g. ms). The "sensor_data!" notion at the transition from and to the "running" state denotes the firing of a "sensor_data" event. The "shutdown?" notion at the transition to the "shutting_down" state means that an event "shutdown" serves as a trigger for the transition. The "not_running" state may be left by the reception of an "acc_ok" event.

The example in Figure 3 models the behavior of the port of component SWC 1, which is connected to the *Input Processing 1* component. This Timed Automaton synchronizes with the first Timed Automaton via the "sensor_data" event. The transition from the starting state to the "processing_data" state is triggered by this event. The processing of the data should happen in the time interval [4, 7] which is annotated in the invariant of the state and the guard of the outgoing transition, which fires the "object_list" event, to which another Timed Automaton will react. A shutting down is modeled just as in the first Timed Automaton.

This small example indicates that there will be a lot of synchronization between many different Timed Automata, especially when a lot of components are connected to each other.

In automotive systems several requirements exist concerning the temporal behavior of the components. In such a case, model checking can be used to verify these requirements.

2.3 Limits of component-based Verification

The verification of component behavior within a component structure has some limits. In distributed systems, the behavior of a specific Timed Automaton usually depends on the behavior of a number of other Timed Automata. In our example we want to verify that a specific state within a specific Timed Automaton has to be reached within a given period

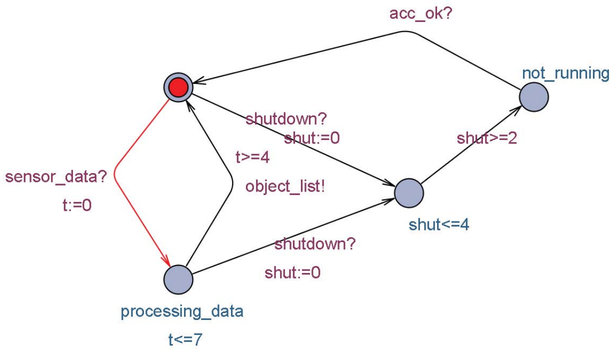


Figure 3: *A second Timed Automaton in Uppaal*

of time. Due to this depends on the global behavior of the whole system, the verification of this constraint is not feasible with the standard mechanisms of Uppaal (Timed Automata and TCTL). The reasons are:

- Every Timed Automaton has its local clock. Global clocks are not available.
- The triggering event and the state, that should be reached in the end, can be in different Timed Automata so that the last automaton has no knowledge about the beginning of the whole process.
- The subset of TCTL that is used in Uppaal is not sufficient to formulate such a formula.

Without a global clock it is not possible to specify all needed TCTL formulas. That means, that it is not possible to specify a constraint in TCTL, in order to verify if a specific state will be reached within a specific period of time. Within Uppaal it is generally possible to define global clocks, but this is a contradiction to the component-based design. So another solution is needed.

2.4 Global Property Verification

As described in the last section, scenarios exist which cannot be verified by Uppaal. So what is missing? Mainly a way to formulate the questions with the functionality provided by model checkers such as Uppaal.

A query, i.e. a question which should be answered by the model checker, consists of two main parts in our approach:

1. A so-called *scenario timed automaton* is used to formulate the query scenario. This scenario mainly comprises the dynamics of the situation-to-be-verified, i.e. the sequence of fired and received events that define the situation-under-test.

   To specify these scenario timed automaton we primarily select states and transitions from the individual Timed Automata, modeled for the component interfaces. Then we combine these selected states and transitions with further states and transitions in order to get a proper Timed Automaton. The result is a scenario timed automaton, which can be used to verify the mentioned constraints.

The scenario timed automaton now provides some kind of global time since its local clock is synchronized via fired and received events with the local clocks of the individual Timed Automaton. This is important because the Timed Automaton associated with the interfaces may only use information that is local to the corresponding software component, i.e. it must be possible to reuse these Timed Automata in other software systems and with other query scenarios. So the use of a global clock within the whole model and as such in each individual Timed Automaton is not desired.

2. Temporal logic formulas are used to define the invariants that should be guaranteed in a given scenario. As outlined above, for timing constraints these formulas may refer to the local clock of the scenario timed automaton.

For the adaptive cruise control example typical scenarios (model queries) are:

- If a new obstacle appears, does the car reach the modified speed within a given time limit?
- If that obstacle disappears, is the optimum speed reached again within a given speed limit?
- Is the active cruise control functionality turned off within a given time limit, if a severe problem is detected within the system?
- If a severe problem occurs within the system, is the driver informed within a given time limit?
- Is the system started within a given time limit?
- Are any deadlocks in the system?
- Many model queries are defined between two specific components, i.e. they check the correctness of one connection. E.g. between the sensor driver component and the next data processing component it makes sense to verify that sensor data is delivered within a given time interval.

A simplified scenario timed automaton for the first query is depicted in Figure 4.

The scenario timed automaton begins in the “SensorData_running” state. If new data is available from the sensors, i.e. if the “sensor_data” event is received, the scenario timed automaton turns to the “start_of_speed_adjustment” state. After the system has adjusted its speed, a “speed_ok” event is received and the system turns to the “VehicleLongitudinalControl_begin” state. This state is a so-called committed state, i.e. it directly switches to the “end” state.

This scenario timed automaton defines the dynamics of one specific scenario. A local clock t is reseted at the beginning which allows temporal formulas to refer to a scenario specific time scale. The committed state at the end is necessary because it allows such

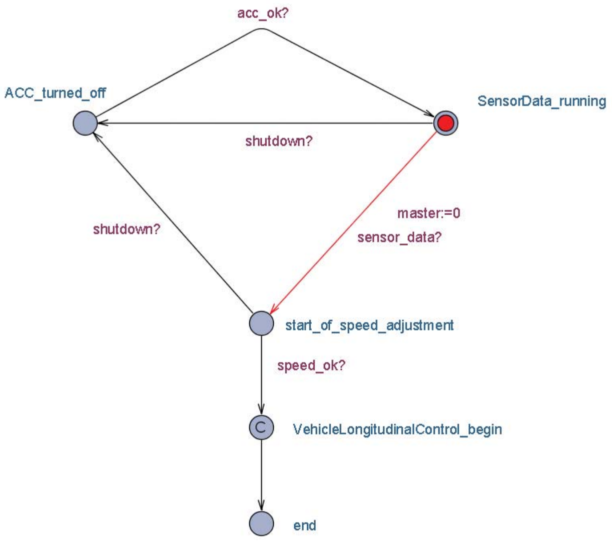


Figure 4: *A scenario timed automaton*

formulas to refer to the point of time when the speed is adjusted, i.e. the state "VehicleLongitudinalControl_begin" is reached.

The temporal formulas that checks for all possible cases whether the speed is adjusted after 10 time units than looks like this:

1. E<> (Scenario1.VehicleLongitudinalControl_begin and Scenario1.t <= 10)

2. E<> (Scenario1.VehicleLongitudinalControl_begin and Scenario1.t > 10)

Scenario1.SensorData_running and Scenario1.VehicleLongitudinalControl_begin refer to the scenario timed automaton from Figure 4. Scenario1.t is the local clock of that scenario timed automaton. Uppaal can now take the Timed Automata associated with ports and connections and the scenario timed automaton for a verification of the temporal formula. The first formula (1) is evaluated to true. The second formula (2) is not fulfilled. That is, the VehicleLongitudinalControl_begin state is entered at least after 10 time units.

3 Conclusion & Future Work

In this paper we presented an approach for the verification of distributed, component-based automotive systems, modeled with Timed Automata. One problem in this context is the appropriate description of the required specification. Therefore, so-called "scenario timed automata" are introduced which enables the engineer to verify more complex requirements.

One of the next steps is to systemize and automate the specification of the scenario timed automata. A mapping from an abstract model to the scenario timed automaton is needed. To support this mapping, an adequate tool-support is needed. Unfortunately Uppaal cannot be used, because within Uppaal it is for example not possible to specify component- or deployment diagrams and associate them with the Timed Automata.

3.1 FUJABA integration

In order to support specifying the architecture of embedded real-time systems which are usually distributed systems, Fujaba[5] has been extended by UML2.0 component diagrams in the Fujaba Real-Time Tool Suite [BGH$^+$05]. The behavior of single components is specified by *Real-Time Statecharts* [GTB$^+$03]. Real-Time Statecharts introduce clocks and allow the specification of worst-case execution times for activities and deadlines for transitions. With the Fujaba Real-Time Tool Suite we are able to transform these Real-Time Statecharts into Timed Automata in order to be used within the model checker Uppaal. One future task is to integrate the idea of the "scenario timed automata" in the Fujaba Real-Time Tool Suite and to extend the available transformation, if necessary.

References

[BGH$^+$05] Sven Burmester, Holger Giese, Martin Hirsch, Daniela Schilling, and Matthias Tichy. The Fujaba Real-Time Tool Suite: Model-Driven Development of Safety-Critical, Real-Time Systems. In *Proc. of the 27th International Conference on Software Engineering (ICSE), St. Louis, Missouri, USA*, May 2005.

[GTB$^+$03] H. Giese, M. Tichy, S. Burmester, W. Schäfer, and S. Flake. Towards the Compositional Verification of Real-Time UML Designs. In *Proc. of the European Software Engineering Conference (ESEC), Helsinki, Finland*, pages 38–47. ACM Press, September 2003.

[OMG03] OMG. *UML 2.0 Superstructure Specification*. Object Management Group, 2003. Document ptc/03-08-02.

[Rob03] Robert Bosch GmbH. *ACC Adaptive Cruise Control*, 2003.

[5] `http://www.fujaba.de`

Enhanced Requirements-Based Programming for Embedded Systems Design

Michael G. Hinchey[1], Tiziana Margaria[2], James L. Rash[1],
Christopher A. Rouff[3], Bernhard Steffen[4]

Abstract: R2D2C is a technique to mechanically transform system requirements via provably equivalent models to running code. In this paper we complement its CSP-based, syntax-oriented model construction, which requires the support of a theorem prover, by model extrapolation via automata learning. The main practical impact of this approach is its power to support the systematic *completion* of the requirements, which by their nature are typically very partial and concentrate on the most prominent scenarios. Our technique generalizes these typical requirement skeletons by extrapolation and it indicates by means of automatically generated traces where the requirement specification is too loose and additional information is required.

1 Motivation

A formal approach to Requirements-Based Programming, provisionally named R2D2C ("Requirements to Design to Code"), was developed at NASA [13] as a general-purpose method to mechanically transform system requirements into a provably equivalent model. This is a central need for ultra-high dependability systems like those developed at NASA for space exploration. The R2D2C approach provides mathematically tractable round-trip engineering for system development, rigorously based on formal modelling and formal reasoning techniques.

In this paper we complement its CSP-based, syntax-oriented model construction, which requires the support of a theorem prover, by model extrapolation via automata learning. The main practical impact of this approach is its power to support the systematic *completion* of the requirements, which by their nature are typically very partial and concentrate on the most prominent scenarios. Our technique generalizes these typical requirement skeletons by extrapolation and it indicates by means of automatically generated traces where the requirement specification is too loose and additional information is required.

Although the approach is not exclusively targeted for embedded systems, it happens to be a central enabler for modelling and analyzing significant portions of space-based systems, which are embedded. Here we briefly sketch two significative application domains.

Application Areas

This work is motivated by the need for requirements-based programming for *ultra-high dependability* systems which are *remote* and *embedded.*

Sensor Networks An example of a sensor network for solar system exploration is the Autonomous Nano Technology Swarm mission (ANTS) [7], which is at the concept development phase. This mission will send 1,000 pico-class (approximately 1 kg) spacecraft to explore the asteroid belt. The ANTS spacecraft will act as a sensor network making observations of asteroids and analyzing their composition. Embedded sensors in space applications are a challenge along several research dimensions: long communications delays with Earth, out of touch with the Earth and mission control for long periods of time, operative under extremes of dynamic environmental conditions.

Due to the complexity of these systems as well as their distributed and parallel nature, they will have an extremely large state space and will be impossible to test completely using traditional testing techniques. R2D2C helps by converting the scenarios into a formal model that can be analyzed for concurrency-related errors, consistency and completeness, as well as domain-specific errors.

Robotic Operations We have been experimenting with generating code to control robots, but more interesting is the use of this approach to investigate the validity and correctness of procedures for complex robotic assembly or repair tasks in space, which rely heavily on the support of embedded controllers. Exploratory work concerns here providing an additional means to validate procedures from the Hubble Robotic Servicing Mission (HRSM) – for example, the procedures for replacement of cameras on the Hubble Space Telescope (HST).

Communication Systems The learning based approaches have fared quite promisingly for the test-based discovery of models of legacy communication systems, thus outperforming prior approaches based on trace combination [10]. As shown in [15, 17], the test-based model generation by classical automata learning is very expensive. It requires an impractically large number of queries to the system, each of which must be implemented as a system-level test case. Key towards the tractability of observation based model generation are powerful optimizations exploiting different kinds of expert knowledge in order to drastically reduce the number of required queries, and thus the testing effort. Recent studies have brought to a thorough experimental analysis of the second-order effects between such optimizations in order to maximize their combined impact [17], and to the development of a mature toolset for experimentation [23], which is used here. As shown in [2], our learning method is coherent with the usual notions of conformance testing.

In the specific R2D2C context, we are interested in investigating the possible application of the combined approach to the specification of communication mechanisms described in the previous application domains. This can be completed by a test-based or monitoring-based validation once those systems are operational.

In the following, we sketch the principles on which the R2D2C approach works and the effects of the learning-enhanced method.

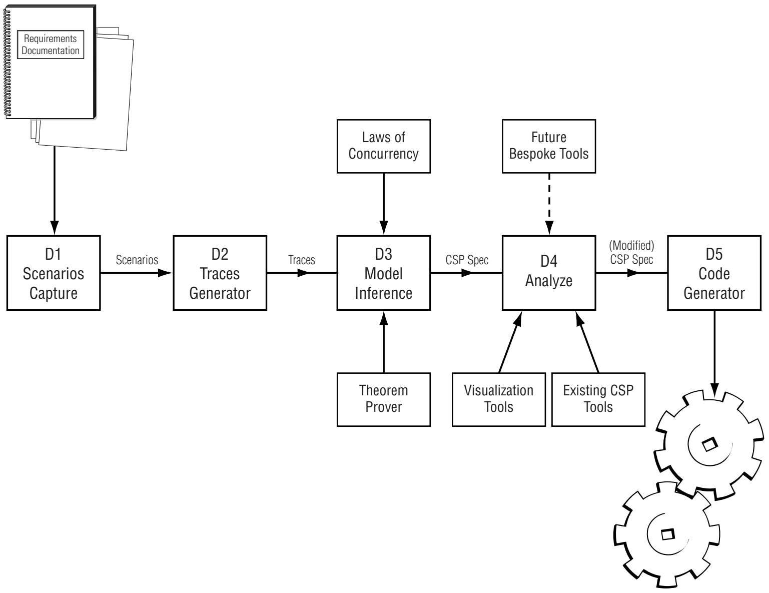


Figure 1: The pure R2D2C approach

2 How R2D2C Works

The R2D2C approach involves a number of phases, which are reflected in the system architecture described in Figure 1 and described below.

D1 Scenarios Capture: Engineers, end users, and others write scenarios describing intended system operation. The input scenarios may be represented in a constrained natural language using a syntax-directed editor, or may be represented in other textual or graphical forms.

D2 Traces Generation: Traces and sequences of atomic events are derived from the scenarios defined in D1.

D3 Model Inference: A formal model, or formal specification, expressed in CSP is inferred by an automatic theorem prover – in this case, ACL2 [16] – using the traces derived in phase 2. A deep[1] embedding of the laws of concurrency [12] in the theorem prover gives it sufficient knowledge of concurrency and of CSP to perform the inference. The embedding will be the topic of a future paper.

D4 Analysis: Based on the formal model, various analyses can be performed, using currently available commercial or public domain tools, and specialized tools that are

[1] "Deep" in the sense that the embedding is semantic rather than merely syntactic.

planned for development. Because of the nature of CSP, the model may be analyzed at different levels of abstraction using a variety of possible implementation environments. This will be the subject of a future paper.

D5 Code Generation: The techniques of automatic code generation from a suitable model are reasonably well understood. The present modeling approach is suitable for the application of existing code generation techniques, whether using a tool specifically developed for the purpose, or existing tools such as FDR [9], or converting to other notations suitable for code generation (e.g., converting CSP to B [4] and then using the code generating capabilities of the B Toolkit).

According to this full cycle, developing a system that will have a high level of reliability requires the developer to represent the system as a formal model that can be proven to be correct. Through the use of currently available tools, the model can then be automatically transformed into code with minimal or no human intervention to reduce the chance of inadvertent insertion of errors by developers. Automatically producing the formal model from customer requirements would further reduce the chance of human error insertion.

In this paper we focus on a specific, new aspect of the R2D2C approach, the completion of the requirements given as a set of traces as generated by D2. This needs a short introduction into automata learning.

3 Automata Learning

Machine learning deals in general with the problem how to automatically generate a system's description. Besides the synthesis of static soft- and hardware properties, in particular invariants [8], [20], [3], the field of *automata learning* is of particular interest for soft- and hardware engineering [6], [19], [25], [21], [5].

Automata learning tries to construct a deterministic finite automaton (see below) that matches the behavior of a given target automaton on the basis of observations of the target automaton and perhaps some further information on its internal structure. [10, 24, 18] explain our view on the use of learning. Here we only summarize the basic aspects of our realization, which is based on Angluin's learning algorithm L^* from [1].

L^*, also referred to as an *active* learning algorithm, learns a finite automaton by *actively* posing *membership* queries and *equivalence* queries to that automaton in order to extract behavioral information, and refining successively an own hypothesis automaton based on the answers. A membership query tests whether a string (a potential run) is contained in the target automaton's language (its set of runs), and an equivalence query compares the hypothesis automaton with the target automaton for language equivalence, in order to determine whether the learning procedure was (already) successfully completed and the experimentation can be terminated.

In its basic form, L* starts with the one state hypothesis automaton that treats all words over the considered alphabet (of elementary observations) alike and refines this automaton on the basis of query results iterating two steps. Here, the dual way of how L* characterizes (and distinguishes) states is central:

- from *below*, by words reaching them. This characterization is too fine, as different words may well lead to the same state.
- from *above*, by their future behavior wrt. a dynamically increasing set of words. These future behaviors are essentially bit vectors, where a '1' means that the corresponding word of the set is guaranteed to lead to an accepting state and a '0' captures the complement. This characterization is typically too coarse, as the considered sets of words are typically rather small.

The second characterization directly defines the hypothesis automata: each occurring bit vector corresponds to one state in the hypothesis automaton.

The initial hypothesis automaton is characterized by the outcome of the membership query for the empty observation. Thus it accepts any word in case the empty word is in the language, and no state otherwise. Now the learning procedure

1. iteratively establishes local consistency after which it
2. checks for global consistency.

Local Consistency: This first step (also called automatic *model completion*) again iterates two phases: one for checking wether the constructed automaton is *closed* under the one-step transitions, i.e., each transition from each state of the hypothesis automaton ends in a well defined state of this very automaton. And one for checking *consistency* according to the bit vectors characterizing the future behavior as explained above, i.e., whether all reaching words with an identical characterization from above possess the same one step transitions. If this is not the case, a distinguishing transition is taken as an additional distinguishing future in order to resolve the inconsistency, i.e., the two reaching words with different transition potential are no longer considered to represent the same state.

Global Equivalence: After local consistency has been established, an equivalence query checks whether the language of the hypothesis automaton coincides with the language of the target automaton. If this is true, the learning procedure successfully terminates. Otherwise the equivalence query returns a counterexample, i.e., a word which distinguishes the hypothesis and the target automaton. This counterexample gives rise to a new cycle of modifying the hypothesis automaton and starting the next iteration.

In any practical attempt of learning legacy systems, equivalence tests can only be approximated (what we will not explain here), but membership queries can be often answered by testing [10, 24]. However, this is no issue for the application scenario we are developing here, as our requirements completion method mainly hinges on the (optimized) local consistency mechanisms.

4 Learning-Based Requirements Completion

Requirement specifications in terms of individual traces are by their nature very partial and represent only the most prominent situations. This partiality is one of the major problems in requirement engineering. It often causes errors in the system design that are difficult to fix. Thus techniques for systematically completing such partial requirement specifications are of major practical importance.

We therefore propose a method for requirements completion, which is based on automatic (active) automata learning. In essence, the method works by

- *initializing* the learning algorithm with the set of traces constituting the requirement specifications, and
- constructing a *consistent behavioral model* by establishing the local consistency introduced in the previous section.

In this fashion, we arrive at a finite state behavioral model, which is an *extrapolation* of the given requirement specification: it comprises *all* traces of the specification, but will typically contain many (even infinitely many) more.

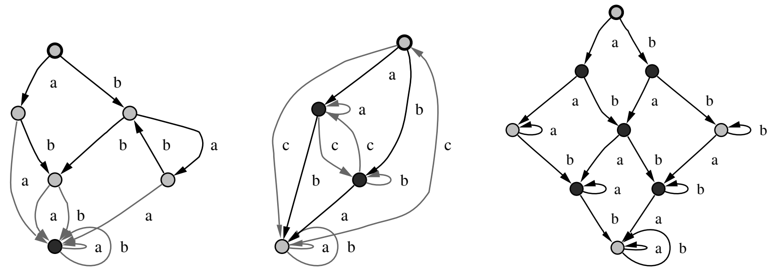
In order for this method to work, a number of membership queries need to be answered. Both, establishing closure of the model, as well as establishing the consistency of the abstraction of reaching words into states (i.e., of the characterization from above introduced in the previous section) can only be effected on the basis of additional information about the intended/unknown system.

This is not unexpected. Rather, it is desired (at least to some extent): the posed membership queries directly hint at the places where the given requirement specification is partial. On the other hand, it is not practical: the number of such membership queries constitute the major bottleneck of active learning, even in the case where it is fully automated. For the approach to membership queries, which by its nature interactive, this is unacceptable.

Our answer to this problem is based on the observation that the number of membership queries can be drastically reduced on the basis of orthogonally given expert knowledge about the intended/unknown system. We could show that already the following three very general structural criteria, *prefix closure*, *independence* of actions and *symmetry*, were sufficient to reduce the number of membership queries by several orders of magnitude. By complementing these optimizations with filters for membership queries based on additional requirement specifications in terms of temporal properties, we reduce the required interaction to a practical level in our first experiments:

Prefix Closure. The set of traces describing the potential runs of a reactive system is by definition prefix-closed: we cannot observe a run of a system without observing all its prefixes. This observation leads to quite dramatic savings concerning membership queries.

Independence of Actions. Components of distributed systems typically proceed independently to a large extent: a lot of their individual actions do not depend on each other and



(a) Prefix Closed DFA (b) Independent Actions a, b (c) Symmetric Actions a, b

Figure 2: Deterministic Finite State Machines with Different Characteristics

can therefore be executed in arbitrary order. This allows us to complete the inferred model by adding all admissible re-shufflings. Depending on the nature of considered systems, this optimization may have a major impact.

Symmetry. Systems comprising sets of identical subsystems, like a set of identical storage cells, or a set of identical autonomous robots, can profit from a symbolic treatment of their subsystems. The symbolic treatment does not work on the individuals, but on the particular roles played by some of them during a particular execution trace: rather than speaking about robot 123, the symbolic treatment speaks about the first robot starting to interact, or the third robot reaching a certain area. For systems with large sets of such identical subsystems, the symbolic treatment turns out to be a must.

Temporal Requirement Specifications. Besides some typical example runs, application experts usually are also able to formulate many necessary safety conditions, e.g. on the basis of required protocols, or the exclusion of catastrophic states. By adding such safety requirements in terms of temporal logics to our requirement specification it is possible to automatically answer a huge number of membership queries by model checking.

We now describe how this technique is embedded into the R2D2C approach.

5 Requirement Completion in R2D2C

Fig. 3 shows the R2D2C scenario including the new requirement completion components. As indicated by the arrows representing the potential flow of R2D2C processes, our new components introduce the following new options, which complement the original R2D2C process here indicated by the arrow bypassing the requirements completion module L2:

- Most powerful is the integrated mode of use, where the requirement completion component L2 is added to the original process. Its role is here simply to support

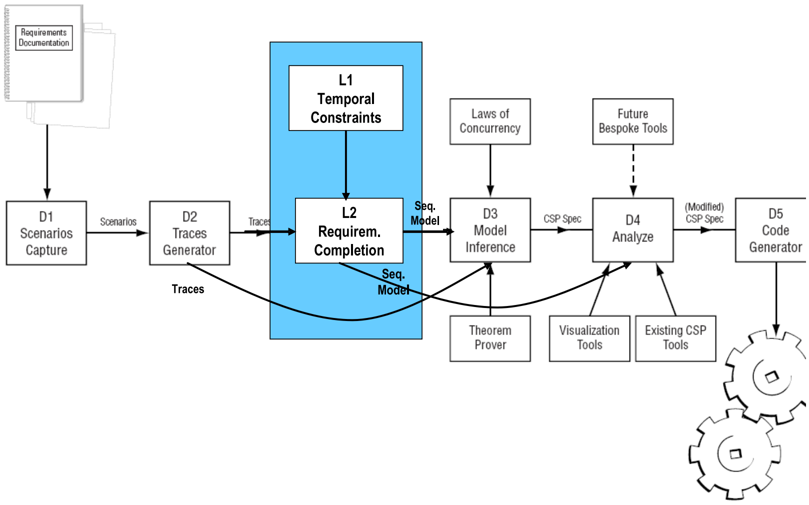


Figure 3: The enhanced R2D2C Approach with Requirement Completion

the evaluation of the given set of requirement traces, and to hint at underspecified portions which may be successively completed. This option strengthens the original R2D2C process.

- Alternatively, one may replace the model inference component D3 by our requirements completion component L2, meaning that the subsequent component D4 and D5 directly work on the model produced by L2. Currently, this means that we restrict ourselves to sequential models. However, we are investigating how to overcome this restriction in the future.

Related Work

Concerning the pure R2D2C approach, which follows the correctness by construction paradigm, several approaches that exploit the maturity of CSP-related tools have already been proposed before. [14] proposes linking the Box Structure Development Method and CSP, with the aim of helping developers familiar with BSDM gain confidence in their artifacts through verification via the FDR model checker of the CSP model produced along the way. Earlier on, the whole ProCoS [22] project had worked out a refinement-based development method that reached from specifications in Duration Calculus to code using CSP as an intermediate modelling language. While the ProCoS stack turned out to be thoroughly rigorous, but difficult to apply to real-life industrial systems, the first approach is used successfully for embedded systems, e.g. in projects for automotive applications. However, the method is applied by the authors themselves, who act as consultants.

Our aim is on the contrary to provide an environment which can be controlled without any knowledge of formal methods. It is meant to steer a dialogue with the requirement engineer (in terms of traces) on his way to a consistent model. Here our learning-based extrapolation approach trades correctness for (faster) convergence (in fact, a more conservative approach would fail to converge for systems with potentially infinitely many runs!). This is also the main difference to other scenario-based play and capture tools, like the play-in play-out approach of [11].

6 Conclusions and Perspectives

R2D2C is a technique to mechanically transform system requirements via provably equivalent models to running code. In this paper we have complemented the CSP-based, syntax-oriented model construction of the R2D2C method with a learning based method for requirement completion. Using automatic (active) automata learning, it is now possible to systematically enrich requirement specifications posed in terms of traces. Key to the practicality of this approach were four optimizations, which limit the number of required user interactions. Whereas the impact of the first three of these optimization has been already discussed earlier, the fourth optimization, which is based on user-given safety specification, is novel. First experimentations are very promising, and indicate that our method may have a significant practical impact.

Currently, we are focussing on the reactive behavior of a system. However, it is possible to also provide (complex) data and real time information with the traces: this is the basis for allowing us to extrapolate models with more complex data and real time properties. The main problem we expect here concerns scalability.

References

[1] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 2(75):87–106, 1987.

[2] T. Berg, O. Grinchtein, B. Jonsson, M. Leucker, H. Raffelt, B. Steffen: *On the Correspondence Between Conformance Testing and Regular Inference*, Proc. FASE 2005, 8th Int. Conf. on Fundamental Approaches to Software Engineering, Edinburgh, UK, April 2005, LNCS N.3442, pp. 175-189, Springer Verlag, 2005.

[3] Y. Brun, M. D. Ernst. Finding latent code errors via machine learning over program executions Proc. 26th *Int. Conf. on Software Engineering* (ICSE'04), pp. 480–490, May 2004

[4] M. J. Butler. *csp2B : A Practical Approach To Combining CSP and B*. Declarative Systems and Software Engineering Group, Department of Electronics and Computer Science, University of Southampton, February 1999.

[5] J. E. Cook, Z. Du, C. Liu, A. L. Wolf. Discovering Models of Behavior for Concurrent Systems Tech. rep. New Mexico State University, Dept. of Computer Science, Aug. 2002

[6] J. E. Cook, A. L. Wolf Discovering Models of Software Processes from Event-Based Data ACM Trans. on Software Engineering and Methodology (TOSEM) pp. 215-249, 1998

[7] S. A. Curtis, J. Mica, J. Nuth, G. Marr, M. L. Rilee, and M. K. Bhat. ANTS (Autonomous Nano-Technology Swarm): An artificial intelligence approach to Asteroid Belt resource exploration. In *Proc. Int'l Astronautical Federation, 51st Congress*, October 2000.

[8] M. D. Ernst, A. Czeisler, W. G. Griswold, D. Notkin. Quickly detecting relevant program invariants In proceedings of the 22nd *International Conference on Software Engineering* (ICSE 2000), 449–458, June 2000.

[9] *Failures-Divergences Refinement: User Manual and Tutorial*. Formal Systems (Europe), Ltd., 1999.

[10] A. Hagerer, H. Hungar, O. Niese, and B. Steffen. Model Generation by Moderated Regular Extrapolation. *Proc. of the 5th Int. Conf. on Fundamental Approaches to Software Engineering (FASE 2002)*, LNCS 2306, pp. 80-95.

[11] D. Harel, H. Kugler, G. Weiss: *Some Methodological Observations Resulting from Experience Using LSCs and the Play-In/Play-Out Approach*, in "Scenarios: Models, Transformations and Tools: International Workshop", Dagstuhl, Sept. 2003. LNCS 3466, pp. 26-41.

[12] M. G. Hinchey and S. A. Jarvis. *Concurrent Systems: Formal Development in CSP*. International Series in Software Engineering. McGraw-Hill International, London, UK, 1995.

[13] Michael G. Hinchey, James L. Rash, Christopher A. Rouff: *A Formal Approach to Requirements-Based Programming,* Proc. ECBS 2005, 12th IEEE Int. Conf. on the Engineering of Computer-Based Systems, Greenbelt (MD), 2005, IEEE, pp. 339-345.

[14] P. Hopcroft, G. Broadfoot. *Combining the Box Structure Development Method and CSP*, ASE'04, 19th IEEE Int. Conf. on Automated Software Engineering, 2004, pp. 340-345.

[15] H. Hungar, T. Margaria, B. Steffen: *Test-Based Model Generation for Legacy Systems*, IEEE International Test Conference (ITC), Charlotte, NC, September 30 - October 2, 2003.

[16] M. Kaufmann and Panagiotis Manolios and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Advances in Formal Methods Series. Kluwer Academic Publishers, Boston, 2000.

[17] T. Margaria, H. Raffelt, B. Steffen: *Analyzing Second-Order Effects Between Optimizations for System-Level Test-Based Model Generation*, Proc. IEEE International Test Conference (ITC), Austin, TX (USA), November 8 - 10, 2005, IEEE Computer Society Press.

[18] T. Margaria, O. Niese, H. Raffelt, and B. Steffen Efficient Test-based Model Generationfor Legacy Reactive Systems. To appear in Proceedings of International High Level Design Validation and Test Workshop, 2004 Sonoma, California.

[19] L. Mariani, Mauro Pezzè. A technique for verifying component-based software Proceeding of the *Int. Workshop on Test and Analysis of Component Based Systems*, TACOS 2004, Barcelona, March 2004

[20] J. W. Nimmer, M. D. Ernst. Automatic generation of program specifications In Proceedings of the 2002 *International Symposium on Software Testing and Analysis* (ISSTA 2002), 232–242, July 2002

[21] D. Peled, M. Y. Vardi, M. Yannakakis Black Box Checking Formal Methods for Protocol Engineering and Distributed Systems, (FORTE/PSTV), pp. 225-240, 1999, Kluwer.

[22] ProCoS Working Group Homepage:
`http://archive.comlab.ox.ac.uk/procos/procos-wg.htm`

[23] H. Raffelt, B. Steffen, T. Berg: *LearnLib: A Library for Automata Learning and Experimentation*, Proc. FMICS 2005, 10th ACM Workshop on Formal Methods for Industrial Critical Systems, Lisbon, Sept. 2005.

[24] B. Steffen and H. Hungar, Behavior-based model construction. In S. Mukhopadhyay and L. Zuck, editors, *Proc. 4th Int. Conf. on Verification, Model Checking and Abstract Interpretation*, LNCS 2575, Springer 2003.

[25] T. Xie, D. Notkin Mutually Enhancing Test Generation and Specification Inference. In Proceedings of 3rd *International Workshop on Formal Approaches to Testing of Software* (FATES 2003), LNCS Vol. 2931, Springer, pp. 60–69, Oct. 2003.

Comparing Heuristics for Model Based Testsuite Generation

Michaela Huhn and Tilo Mücke
Technical University of Braunschweig,
38106 Braunschweig, Germany
{huhn,tmuecke}@ips.cs.tu-bs.de, URL: http://www.cs.tu-bs.de/ips

Abstract: The model driven approach has been applied successfully not only in system development but also in validation and verification. Automated test case generation from specification and design models has attracted wide attention, in particular for embedded, reactive applications. In case adequately formalised semantics for the models is provided, model checking or other search techniques can be used to automatically derive test cases, which are often complex interaction sequences between the system and its environment.

We give an overview on our recent work on optimised test case generation for state based models. First we show how to instrument the models to achieve different test quality criteria like testing particular scenarios or properties, coverage criteria and mutation testing. Then we compare a number of heuristics to reduce test execution time of a testsuite but retain the required test quality. Finally, we take up a recent discussion on the fault detection rate of automatically generated test cases and discuss the implications for our different search and optimisation heuristics.

1 Introduction

In the embedded domain, model based software development fits very well in the established systems engineering processes [VDI04] since tool supported modeling and simulation has a long tradition in mechanical and electrical engineering. To formally verify the software components within such systems, models abstracting from implementation details are also highly recommended to cope with the size and intrinsic complexity of many embedded applications. In the last decade, a number of authors proposed the use of model checkers [EFM97, RH01, HLSC01] and related search techniques [Pre01] from the formal verification area for the derivation of test cases from design and specification models. At first, these approaches to automated test case generation aims to reduce the time and effort spent in testing which is nowadays a predominant cost factor in system development. Additionally, the testsuite generation procedure can be optimised with respect to cost functions like the test execution time [MH04, HLN$^+$03].

However, recently the fault detection rate of automated testsuites generated for structural coverage criteria on models has been questioned: Heimdahl [HDW04] observed weaknesses because in some case studies the generated testsuite contained mainly short test cases that reached only a small part of the state space.

We follow this line of discussion and present a uniform approach to instrument deterministic state based models such that testsuites for different test quality criteria like functional test purposes, structural coverage criteria or mutant testing can be generated by model checking. Moreover, we consider optimisation heuristics for testsuites that reduce the test execution time but retain the test quality criterion. With respect to the fault detection rate the improvements of our approach are twofold: Our test quality criteria are more diverse than pure structural coverage criteria. Moreover, our heuristics for testsuite optimisation substantially enlengthen the test cases which solves a major weakness observed by Heimdahl and others. Last but not least, we add so called *unique input output sequences* (UIOS) that ensure that the system under test has indeed reached the correct state after executing a test sequence. If a test case reaches a fault, the UIOS guarantees that it is uncovered if the fault is reflected in the system configuration. This is an additional strong measure to improve fault detection.

2 Test Case Generation

2.1 Overview

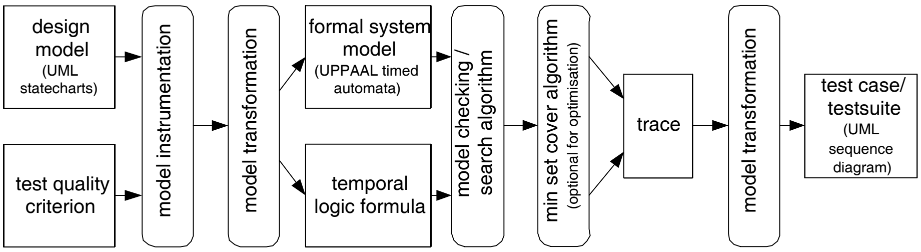

design model (UML statecharts)
test quality criterion
model instrumentation
model transformation
formal system model (UPPAAL timed automata)
temporal logic formula
model checking / search algorithm
min set cover algorithm (optional for optimisation)
trace
model transformation
test case/ testsuite (UML sequence diagram)

Figure 1: Automated model based test case generation

Figure 1 summarises the test case generation procedure. Initially, the model is instrumented according to the selected test quality criterion. The instrumentations needed for the different test quality criteria are described in the next subsection. For our purposes, systems are modelled as a family of UML statecharts [OMG03] describing the behaviour of the system components. There exist a number of formal semantics for statechart dialects and we will use the approach from [DGH02, MH04] to transform statecharts into the input language of the UPPAAL model checker [LPY97] and translate the output traces of the model checker back into sequence diagrams. The UPPAAL model checker was chosen because we are particularly interested in the timing behaviour, which is modelled by real-time annotations at transitions and states. We consider different search strategies for test case generation that are based on the idea that redundant testsuites with long test cases have larger optimisation potential. After the generation phase, the testsuite is reduced by a min-set-cover algorithm that ensures the required test quality is retained.

2.2 Instrumenting the Models

We consider the instrumentation of models for test case generation via model checking for three different test quality criteria:

1. A *test purpose* is given by a test expert. It consists of a desired property or a critical operation sequence. Test cases checking the property or executing the sequence are generated. Which types and amount of faults are found depends on the purpose specification given by the test expert.
2. *Coverage criteria* are definitions of model element type dependent, structural properties which have to take place during test case execution. E.g. state coverage demands each state to be visited at least once.
3. *Mutant testing* demands that every mutated model has to be uncovered as erroneous by the testsuite, unless it behaves equivalent to the original model. The mutants are generated automatically by so called mutation operators.

A theoretical comparison of the fault detection rates of coverage criteria and mutant testing can be found in [MG05] and asserts that both techniques are equally potent.

We decompose each test quality criterion in subgoals which are to be achieved and call them *partial coverages*. A partial coverage consists of some behaviour and a state meaning that this behaviour should occur on some execution starting from that state. For instance "*a sequence has to be observed starting from state s*" or "*transition t has to be reached (from the initial state)*".

The key idea to instrument the design models for test case generation according to a partial coverage is to add monitoring facilities which observe the behaviour to be tested. If the behaviour has occurred a specific *success* predicate is set. Now the model checker searches for a sequence starting at the required state and leading to *success*. In the simplest case of state coverage, the monitor is just the state to be reached itself. In more complex cases, auxiliary variables, refinements of the models or additional statecharts encoding scenarios or mutated models are used for monitoring.

At first, a test driver is added which feeds the system under test with all possible inputs. The test driver is implemented nondeterministically. Technically, the test driver is put in parallel to the statecharts modeling the system which leads to a product construction at the level of model checking. Fortunately, the test driver consists of only one state.

Test Purposes

Test Purpose: Property. The simplest case is some property that is already expressible in terms of the design model. Then a test case can be generated by giving the model checker the corresponding predicate and an initial state. The result will be a test case (sequence) to some state where the desired property holds, if possible.

Test Purpose: Interaction Sequence. In many cases a test purpose can be naturally described as a sequence of operations or interactions. As many others, we use sequence diagrams to represent such interaction sequences. To generate a test case including this sequence, the test driver is extended by a monitor automaton observing the desired sequence. Here we use constructions known from the transformation of scenario based models into state based models extended by the handling of timing constraints.

Structural Coverage Criteria.

A structural coverage criteria requires that all occurrences of a particular model element or other situations that can be described on the syntactic structure of the model are reached in test cases. To generate test cases for structural coverage criteria, the models are instrumented by monitor variables that are set when the required model element is used.

State coverage. In case a testsuite for state coverages shall be generated, each reachable state s describes a partial coverage, namely that a test sequence is needed that leads from the initial state to s.

Transition Coverage. To achieve this coverage each transition has to fire in at least one test case. The model is instrumented by adding a boolean coverage variable for each transition which is set to true, if the associated transition fires.

Data Flow Coverage. Data flow coverage requires test cases for every path from an assignment of a variable to its subsequent usage. A number of variants of data flow criteria are known from test generation on the code level. They are be handled by adding auxiliary variables monitoring each assignment-usage path of a variable, similarly to transition coverage.

Boundary Coverage. This coverage criterion demands that for each guard containing a relational operator a test case is generated for which the operands are as close as possible to the boundary. To achieve this coverage the model is refined by splitting the transitions containing a guard with relational operators: One transition is labelled with a guard capturing the exact boundary and monitored by a coverage variable and another one preserves the original behaviour in all other possibilities.

Other coverage criteria are treated similarly.

2.3 Mutant Testing

Mutant testing requires that each mutation of the original model is either equivalent to the original model or found as erroneous by the testsuite. The mutants are generated by mutation operators.

The classical mutation operators [KO91] are applied to code only. Thus, in our case, we can apply them to guards and actions within the statecharts. Common mutation operators are: (1) LCR, AOR, ROR which replace each occurrence of a logical, arithmetical, or relational operator by any other operator of the same type, (2) UIO which negates, increments and decrements each arithmetic expression, (3) AAR, ACR, ASR, ... replace each occurrence of a variable, constant or array by each compatible variable, constant or array,

(4) CRP, DSA slightly change constant values, (5) SDL deletes each statement, and many more.

On the model-layer, these mutation operators have to be supplemented by mutation operators working with the structure of the automata [SMW04]. Some of these operators are already handled by mutating guards and actions. Examples for other operators are: (1) state missing, (2) transition missing, (3) replace origin state of a transition, (4) replace target state of a transition, (5) replace triggering event, (6) replace triggered event, and (7) replace event recipient.

To generate mutant killing test cases, the original model has to be executed versus a mutated one. Both are triggered by the same inputs. A mutant is found as erroneous, if the outputs differ. Technically, a test case is generated by model checking the original and the mutated system in parallel. A comparator compares the outputs of the systems, announcing *success* whenever the two systems produce different output.

Even if we assume the original model to be deterministic, the generated mutants for structural mutation operators might be nondeterministic, for instance for the structural operators (3) and (5). This leads to generated test cases which do not ensure that the mutants are found, because the mutation can only be detected if one particular choice is taken. Fortunately, adding operator (2) solves this problem. It also checks, if the transition still leaves the same state triggered by the same event and therefore the non-deterministic part of the change which happens through operands (3) and (5).

3 Strategies to Generate Time Optimized Testsuites

After instrumenting the model according to a test quality criterion, the statecharts modeling the system under development are transformed into a semantically equivalent family of timed automata that serves as formal system model for UPPAAL. Details on the construction, e.g. syntactic restrictions on the UML model elements, the translation of timing constructs, the handling of event queues and UML run-to-completion semantics, can be found in [DGH02, MH04]. In earlier work [MH04] we showed how to employ the UPPAAL model checker to search for the optimal testsuite with respect to test execution time. But this approach is restricted to small case studies due to the memory consumption in the generation phase.

Alternatively, we consider heuristics to efficiently generate test suites with optimized but not necessarily the minimal execution time. The procedure works in two steps:

3.1 Generating a Redundant Testsuite

In the first step we generate a testsuite containing test cases for all partial coverages that arer required by a test quality criterion. For this generation phase, we have investigated several search heuristics:

Naive Search. We generate a test case for each partial coverage we are interested in, thereby following the work of Hong et.al. [HLSC01]. These test cases build a testsuite for a given test quality criterion as each required partial coverage is achieved at least once. Moreover, some partial coverages may be satisfied by more than one test case[1].

To increase the basis for optimization we enlengthen the test cases by adding a path starting from its final state and leading to a state where some additional coverage is satisfied. The aim is to generate promising long test cases.

Depth 2 search aims to enlengthen a test case by a suffix that achieves an additional partial coverage. The initial parts are generated by the naive search strategy and afterwards we iterate on all partial coverages of interest and extend the test case by a suffix reaching the additional coverage. The number of test cases generated by Depth 2 search is in $O(|PC|^2)$.

Heuristic searches enlengthen only the best test cases for a partial coverage which has been achieved rarely so far. Therefore we add ranking functions that measure the quality of a test case and how often a partial coverage is covered already. For instance, to optimize the test execution time of a testsuite the quality of a testcase is the sum of its timing annotations (its execution time) in proportion to the number of partial coverages it achieves. The amount of redundancy with respect to the achieved coverages is controlled by a parameter of the search.

3.2 Optimizing the Testsuite

We use min-set-cover-algorithms [OPV95, PK89] to optimize test execution time of a testsuite but retain the required partial coverages. Originally, a min(imal)-set-cover algorithm constructs a small subset from a set of sets, such that the union of sets in the small subset equals the union of the sets in the original set. Since the minimal set cover problem is NP-complete [GJ79], heuristic algorithms are used. Since the testsuite has been enlarged, a min-set-cover-algorithm works on a broader basis from which it eliminates redundant test cases with respect to the achieved coverages but with significantly smaller execution time. We consider variants of greedy algorithms:

The simple greedy algorithm tries to eliminate test cases just in the order the test cases occur in the testsuite. Thus we iterate on the test cases an eliminate those that are redundant with respect to the partial coverages they achieve.

Sorted greedy algorithms improve testsuite optimization by selecting the candidates for elimination in a better way: We tried different ranking functions to select weak test cases first for elimination.

A force directed algorithm takes not only an absolute quality measure for the test cases into account but its current value within the testsuite, i.e. the order of candidates for elimination is not determined a priori but depends on the other test cases that are still in the testsuite. Therefore we use a weighted ranking function where the weight gives a measure of the ”uniqueness” of a test case in the testsuite with respect to the partial coverages it

[1] A test case for the coverage c_1 may reach other coverages on the way.

achieves. Thus, a ”mainstream” test case achieving only partial coverages that are covered by several others, too, will be ranked very low.

3.3 Experimental Results

We investigated each combination of a search algorithm for testsuite generation and a min-set-cover algorithm for testsuite optimisation. Experiments were done on case studies (light-controller [MH05b] and middleware [MH04]) and on randomly generated statecharts ([MH05b]). The results show that a good optimization of the test execution time requires both, a clever search for a redundant testsuite *and* an appropriate min-set-cover algorithm. Thereby we achieved a reduction to approx. 5 to 10% of the test execution time compared to a testsuite without time optimization. Moreover, compared to the generation of the time optimal testsuite we observed significant improvements w.r.t. time and memory efficiency.

4 Improving the Fault Detection Rate

As already mentioned, it turned out in some case studies [HDW04] that the fault detection rate of testsuites generated for structural coverage criteria like state or transition coverage is weak. But fault detection is the overall aim of testing. The problem occurs in state spaces where most structural partial coverages are reachable via short test cases. Then, only the initial part of the state space is tested. The first step to solve this problem is to generate test cases for test purposes that specify the regular and the exceptional behaviour of the system or reasonable system properties. Test cases for structural coverages are helpful to fill the gaps, in addition to functional tests of the behaviour. In our approach, the search strategies generating redundant testsuites enlengthen the test cases and we can control the length of the preferred test cases also in our optimisation heuristics. Our experiments show that that the strategies work and optimised testsuites tend to have long test cases. Thus the problem is avoided at least partially even if only structural coverages are considered. Additionally we offer *unique input output sequences* (UIOS) as a mean to enforce that faults that can be detected at some state are indeed uncovered by a test case:

4.1 Unique Input Output Sequences

Another reason for the weakness of a straightforward test case generation procedure for a structural coverage criterion can be easily understood on the example of transition coverage. To achieve transition coverage test cases are generated in which the last step is the execution of the desired transition. Thus, it is not tested if the correct target state of the transition is actually reached. Similar arguments can be given for structural coverages for other model elements. To overcome this kind of problems, we enlengthen the test cases by so called UIOSs (Unique Input Output Sequences). A UIOS consists of a sequence which can only be executed starting at a specific state. From all other states, provided they are not trace equivalent, outputs of the system will differ from the outputs described in the UIOS.

Technically, UIOSs are handled similar to mutant testing: We generate UIOSs using model checking [RMLMG05] by putting several systems with partly modified initial states in parallel. Thus, we consider all systems with ”mutated initial states” simultaneously. A comparator compares the outputs of the systems, changing a monitor variable whenever two systems produce different output. As soon as a system has produced different output in comparison to all other systems, a UIOS for the initial state of this system is found.

A UIOS extends the test cases in the following way: if a partial coverage is reached, immediately one corresponding Unique Input Output Sequence is executed. UIOSs significantly improve the fault detection rate because if the fault results in any effect on the system state it is uncovered by the UIOS, but for the prize that they substantially enlengthen the test cases and thereby increase test execution time.

5 Conclusion

We presented an approach for the automated generation of optimised testsuites for state based models. We considered a catalogue of test quality criteria, namely the test of system properties or interaction sequences, various coverage criteria, and mutant testing. State based models are instrumented by adding variables or specific test drivers such that a model checker searching for a subgoal encoded as partial coverage will generate a trace that can serve as a test case for that subgoal. Thus, not only structural test quality criteria like coverages but also expert knowledge and existing tests in terms of sequence diagrams are integrated.

We propose several heuristics for optimising the test execution time without decreasing the test quality. We combined search algorithms, adding redundancy on the required coverages to a testsuite, with different min-set-cover algorithms that preserve the set of coverages but minimise the execution time.

Our experimental results are promising under three aspects: First, the test execution time could be significantly reduced. Second, the heuristics for optimisation are efficient w.r.t. time and memory consumption such that our approach is applicable on medium sized real world case studies at least. That is an improvement compared to the generation of time optimal testsuites which is restricted to rather small models. Third, our approach favours the generation of long test cases on which several subgoals (partial coverages) are tested. Thus, we avoid known weaknesses w.r.t. the fault detection rate that were observed on other approaches to automated testsuite generation. The technical details of this paper can be found in [MH04, RMLMG05, MH05b, MH05a].

In future, we plan to extend our work on mutant testing for state based models and investigate alternative heuristics for testsuite generation like e.g. genetic algorithms.

References

[DGH02] K. Diethers, U. Goltz, and M. Huhn. Model Checking UML Statecharts with Time. In *UML 2002, Workshop on Critical Systems Development with UML*, 2002.

[EFM97] A. Engels, L. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In *Tools and Algorithms for the Construction and Analysis of Systems*, 1997.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman and Company, 1979.

[HDW04] Mats P.E. Heimdahl, George Devaraj, and Robert J. Weber. Specification Test Coverage Adequacy Criteria = Specification Test Generation Inadequacy Criteria? In *Proceedings of the 8th IEEE International Symposium on High Assurance Systems Engineering (HASE)*, Tampa, Florida, March 2004.

[HLN$^+$03] A. Hessel, K. Larsen, B. Nielsen, P. Pettersson, and A. Skou. Time-Optimal Test Cases for Real-Time Systems. In *Workshop on Formal Modeling and Analysis of Timed Systems*, 2003.

[HLSC01] H. Hong, I. Lee, O. Sokolsky, and S. Cha. Automatic Test Generation from Statecharts Using Model Checking. In *Workshop on Formal Approaches to Testing of Software (FATES)*, pages 15–30, 2001.

[KO91] K. N. King and A. Jefferson Offutt. A Fortran language system for mutation-based software testing. *Software–Practice & Experience*, 21(7):685–718, 1991.

[LPY97] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[MH04] Tilo Mücke and Michaela Huhn. Generation of Optimized Testsuites for UML Statecharts with Time. In Roland Groz and Robert M. Hierons, editors, *TestCom*, volume 2978 of *LNCS*, pages 128–143. Springer, 2004.

[MH05a] Tilo Mücke and Michaela Huhn. Minimizing Test Execution Time During Test Generation, 2005. submitted.

[MH05b] Tilo Mücke and Michaela Huhn. Optimising Test Execution Times in Test Suite Generation. In *Informatik 2005 - Workshop Modellbasierte Qualitätssicherung (QUAM 2005)*, Bonn, Germany, September 2005.

[OMG03] OMG. Unified Modeling Language Specification, 2003. Version 1.5.

[OPV95] Jeff Offutt, Jie Pan, and Jeff Voas. Procedures for Reducing the Size of Coverage-based Test Sets. In *Proceedings of the Twelfth International Conference on Testing Computer Software*, pages 111–123, 1995.

[PK89] P.G. Paulin and J.P. Knight. Force-directed Scheduling for the Behavioural Synthesis of ASICs. *IEEE Trans. on Computer-Aided Design*, 8(6):661–679, 1989.

[Pre01] A. Pretschner. Classical search strategies for test case generation with Constraint Logic Programming. In *Workshop on Formal Approaches to Testing of Software (FATES)*, pages 47–60, 2001.

[RH01] S. Rayadurgan and M. Heimdahl. Coverage Based Test-Case Generation using Model Checkers. In *Intl. Conf. and Workshop on the Engineering of Computer Based Systems*, pages 83–93, 2001.

[RMLMG05] Christopher Robinson-Mallett, Peter Liggesmeyer, Tilo Mücke, and Ursula Goltz. Generating optimal distinguishing sequences with a model checker. In *A-MOST '05: Proceedings of the 1st International Workshop on Advances in Model-based Testing*, pages 1–7, New York, NY, USA, 2005. ACM Press.

[SMW04] Tatiana Sugeta, José Carlos Maldonado, and W. Eric Wong. Mutation Testing Applied to Validate SDL Specifications. In Roland Groz and Robert M. Hierons, editors, *TestCom*, volume 2978 of *LNCS*, pages 193–208. Springer, 2004.

[VDI04] VDI. Design methodology for mechatronic systems, 2004. VDI 2206.

UML-basierte Entwicklung sicherheitskritischer Systeme im Bahnbereich

Hardi Hungar[*]
OFFIS, Oldenburg

hungar@offis.de

Abstract: Es werden Ansatz und Ergebnisse des OpRail-Projektes vorgestellt. Das Projekt hat zum Ziel, mit den CENELEC-Normen konforme Entwicklungsprozesse für sicherheitskritische bahntechnische Anwendungen zu optimieren. Der optimierte Prozess soll besonders die Verwendung formaler Methoden in einem UML-basierten Gesamtprozess im Fokus haben.

Im Bahnbereich ist die Entwicklung sicherheitskritischer Systeme durch die EN-Normen 50126, 50128 und 50129 (füher CENELEC-Normen) geregelt. Ziel der Normen ist, Anforderungen an die Zuverlässigkeit des Systems in Abhängigkeit der Kritikalität zu vorzuschreiben und überprüfbar zu machen, dass das Resultat diese Anforderungen erfüllt. So enthalten die Normen Anweisungen, Regeln und – in weniger gut beherrschten Bereichen – Vorschläge für den Entwicklungsvorgang. Zu den bisher aus praktischer Sicht noch unzureichend fixierten Bereichen gehört insbesondere die Frage des Nachweises der Anforderungen, wenn die Realisierung komplexe digitale Hard- und Software beinhaltet. Traditionelle Methoden der Sicherstellung – vollständige Analyse aller möglichen Zustände – greifen nicht. Andererseits treten formale Methoden mit dem Versprechen auf, just solche vollständigen Zusicherungen geben zu können. Die Normen machen allerdings in dieser Hinsicht noch keine konkreten Vorgaben, sondern listen eine große Anzahl von Verfahren auf, ohne jeweils die Anwendungsbereiche zu präzisieren. Der Gründe sind einfach: Der Sektor ist noch in einer lebhaften Entwicklung und es gibt wenig Erfahrung aus der Praxis, geschweige denn etablierte Verfahren.

An dieser Stelle setzt das Projekt OpRail an. Ziel ist die prototypische Realisierung eines Prozesses, der die normkonforme Verwendung formaler Methoden demonstriert. Auf der Basis des UML-Werkzeuges Rhapsody (i-Logix) wird ein Prozess definiert, der den Normanforderungen Rechnung trägt und an formalen Methoden

- Modelchecken zur funktionalen Verifikation,
- Laufzeitanalysen und
- automatische Testgenerierung

[*] Diese Arbeit wird vom BMBF unter dem Förderkennzeichen 01ISC26A im Rahmen der Forschungsoffensive ”Software Engineering 2006” gefördert.

vorsieht, für die jeweils spezielle Werkzeuge eingesetzt werden. Dieser Prozess wird an zwei industriellen Fallstudien exemplarisch durchgeführt. Ein wesentlicher Bestandteil des Prozesses ist SafeUML, ein Leitfaden zur Auswahl der UML-Konstrukte, die für die Entwicklung sicherheitskritischer Bahnsysteme verwendbar sind.

Im Konsortium sind folgende Organisationen vertreten, wobei jeweils die Rolle im Projekt bezeichnet ist.

- das Oldenburger Forschungsinstitutes OFFIS (Koordination, Safe-UML, Modelchecken, Basisverfahren formaler Methoden)
- die Firma Alcatel (Fallstudie Stellwerkstechnik)
- die Firma Berner & Mattner Systemtechnik AG (Prozess)
- die Firma DEUTA-Werke GmbH (Fallstudie Odometriekomponente)
- das Institut für Eisenbahnwesen und Verkehrssicherung der TU Braunschweig (Modellierung von Bahnsystemen)
- die Firma OSC - Embedded Systems AG (Safe-UML, Automatische Testgenerierung)
- der TÜV Rail (Normkonformität).

Das Projekt befindet sich im letzten Halbjahr der zweieinhalbjährigen Laufzeit. Prozess und Werkzeuge sind in der Erprobung an den Fallstudien. Wenn die Ergebnisse zur Zeit auch noch voräufig sind, lässt sich erkennen, dass bei Berücksichtigung der Grenzen der Methoden und Werkzeuge ein UML-basierte Prozess unter Verwendung formaler Methoden für die Praxis tauglich ist.

An Actor-Oriented Model-Based Design Flow for Systems-on-Chip

Leandro Soares Indrusiak, Manfred Glesner

Microelectronic Systems Institute
Technische Universität Darmstadt
Karlstr. 15
64283 Darmstadt
lsi@mes.tu-darmstadt.de
glesner@mes.tu-darmstadt

Abstract: This paper presented an approach for a model-driven Systems-on-Chip design flow using actor-orientation as the underlying framework for modeling and simulation. Two case studies using Ptolemy II and FPGA-based prototyping platforms are describing, validating specific steps of the proposed flow.

1 Introduction

Systems-on-Chip (SoC) designers are familiar with model-driven design flows, even if they are not sure what this means. Hardware design flows are based on models and model transformations for decades already, so similar methods have been tried out in SoC design flows, where the hardware and software parts of a system are designed together. The major problem is the intrinsic difference between the model-driven approaches of the hardware and the software parts. Regarding synchronization, traditional hardware modeling is based on inherently concurrent components and strict synchronization with clock signals. Software modeling often abstracts concurrency away or deals with it using threads of control. The level of abstraction of the behavior description in hardware and software also differs, as hardware behavior is described using state machines, boolean equations or register transfer level models, while software can express its behavior in programming code itself or, more recently, UML actions and action languages.

This paper addresses the problem of integrating model-driven design flows of the hardware and software parts of a System-on-Chip. The approach presented here is a first attempt to combine multiple levels of abstraction, regarding both behavior description and synchronization among components, in a single modeling environment.

To comply with the requirements of typical SoC flows, the modeling environment must allow the verification and profiling of the models through simulation, and must also allow the modeling of complex testbenches that mimic the real-world deployment scenario of the SoC.

The paper is organized as follows: the next section will provide an overview on actor orientation, which is the framework used to combine multiple models of concurrency in this work. Then, an overview of the proposed SoC design flow is presented and two case studies validating several steps of that flow are given.

2 Actor Orientation

Previously regarded only as design visualization interfaces, block-based visual modeling tools are gaining acceptance within industry and academy for the design of dataflow-dominated Systems-on-Chip in application domains such as digital signal processing (DSP). Examples include design tools such as SystemGenerator from Xilinx, SPW from CoWare, Advanced Design System from Agilent and Synplify DSP from Synplicity, as well as the general purpose simulation platform Simulink from MathWorks. Such tools follow a simple scheme for system modeling, using hierarchical blocks that can be interconnected with other blocks through ports. Rich block libraries are provided, so designers can build complex systems by instantiating and connecting library elements.

The underlying simulation semantics of such environments is often transparent to the users, which concentrate mainly on structuring and parametrizing the system. However, in many cases a particular simulation semantics can facilitate or prevent the efficient and accurate modeling of a given system, specially when it comes to model concurrency [PS04]. To allow for full exploration of different simulation semantics, Lee [LN03] coined the concept of actor orientation and proposed that the simulation semantics should be also part of the model, allowing for complex models using distinct semantics on different subsystem. The components in actor-oriented models are concurrent objects that communicate via messaging, rather than abstract data structures that interact via procedure calls (which is the picture in object-oriented programming languages). A distinct feature of actor-oriented models is the formalization of the concurrent behavior of model components using well defined models of computation (MoC). While many actor-oriented tools like Simulink or LabVIEW rely on a single MoC, the PtolemyII project [Le04] has provided a platform for experimenting with heterogeneous models combining different MoCs. It allows the full exploration of actor-based models in domains other than the dataflow-oriented ones mentioned earlier in this section, like in control-flow oriented systems and event-based systems.

The support to multiple MoCs in a single system model allows also for the coexistence of different abstraction levels, for instance the co-simulation of a physical subsystem that may be represented using continuous time semantics and the digital hardware controlling that physical subsystem, which may be modeled using discrete event semantics.

Our approach to actor-oriented design also aims to take advantage on multiple MoCs for modeling heterogeneous systems as well as their usage scenarios. The design flow will then be based on successive refinements of specific subsystems (either through synthesis, code generation or manual implementation), allowing the refined subsystems to be validated with the complete system model through co-simulation.

3 Design Flow

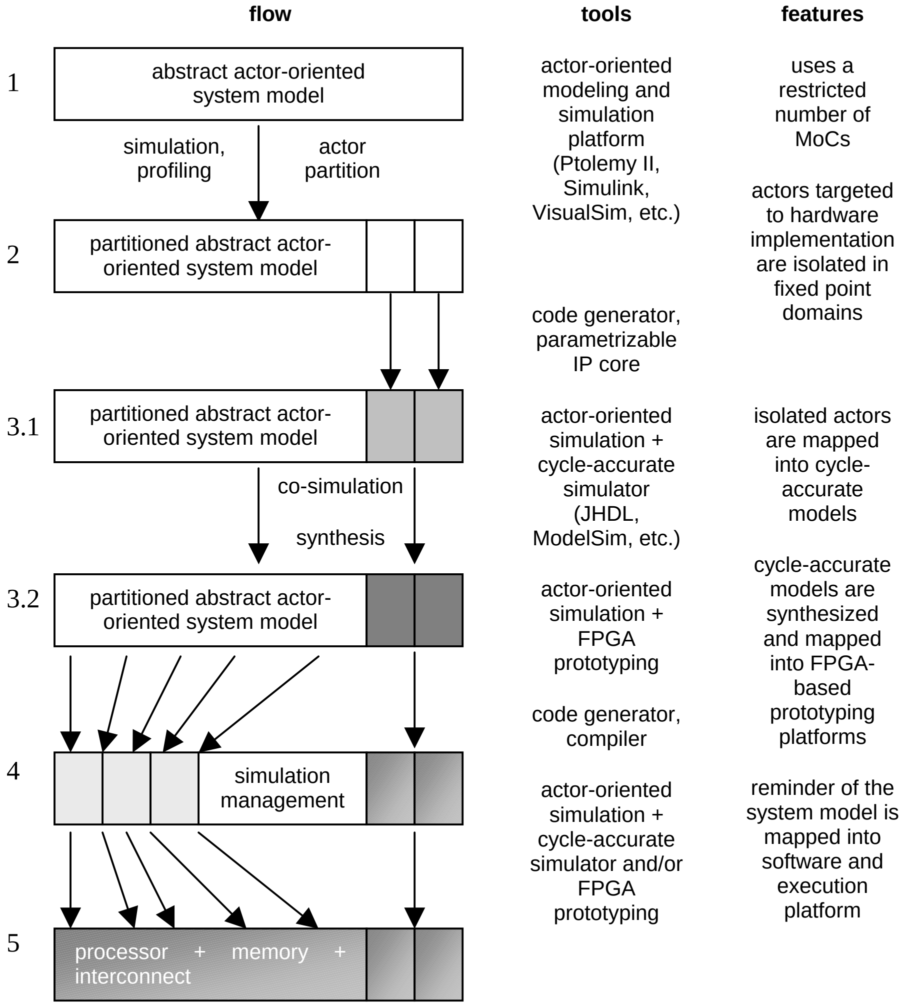
The first step of the proposed design flow comprehends the actor-oriented modeling of the system functionality. This models includes the whole system behavior and incorporates all details of its testbench. For instance, the actor-oriented model of a wireless communication system would include the models of the transceiver – which will later be implemented in hardware and software – and also the models for the wireless channel conditions, user mobility patterns, interferences, multipath effects, etc. The testbenches increase significantly the complexity of the system model, therefore such model is implemented in a high level of abstraction, where the individual system components may not yet be well delimited (an actor at this level may end up being a cluster of actors in a next iteration of the modeling phase).

Once the system is modeled and profiled through simulation, the next step is to isolate the subsystems that show higher performance requirements as they are the best targets to be implemented in hardware. Actor partitioning may be required, and floating-point to fixed-point conversion actors may be used to evaluate the quantization alternatives and its impact in the system performance and quality of results. No automation was used for this step, but we can expect that some automated or semi-automated actor partition techniques could be developed, and that current tool-aided quantization techniques could be adapted to fit here.

The third step comprehends the successive prototyping of the isolated actors into hardware description languages and then in hardware prototyping platforms. Each actor should be first mapped into a cycle-accurate synthesizable model that should be encapsulated as an actor within a discrete-event domain of the system model. Manual conversion, code generation or parametrizable intellectual property cores (IP cores) can be used for the first model transformation. Co-simulation between the less-abstract, cycle-accurate model and the rest of the system model is possible, so the consistency of the first model transformation can be verified. The cycle-accurate model is then synthesized and mapped into an FPGA (Field Programmable Gate Array) based prototyping platform, which can also be encapsulated within an actor and co-simulated with the rest of the system model in a “hardware-in-the-loop” fashion.

Finally, once all targetted-to-hardware actors were prototyped, the remaining actors in the system model should be mapped to software threads running in one or more processors. The exploration of alternatives regarding operating systems, processor architectures and memory organization can start at the system level already, as several actor-oriented tools already include models for different processors, interconnect architectures, memories and operating systems.

Once a given alternative is defined, cycle-accurate models of the software execution platform (processors, memory, interconnects) should be encapsulated within actors as it was described in the third step, and a FPGA-based prototyping can also be done. The co-simulation of the cycle-accurate model and the FPGA-based prototype within the system simulation environment has a single advantage, which is the increased observability and debugability, so this step can be even neglected and the final validation of a fully-cycle-accurate model of the system can be directly done. Figure 1 depicts the proposed design flow, where arrows represent model transformation using the referenced tools.



flow
tools
features
1
abstract actor-oriented system model
simulation, profiling
actor partition
actor-oriented modeling and simulation platform (Ptolemy II, Simulink, VisualSim, etc.)
uses a restricted number of MoCs
actors targeted to hardware implementation are isolated in fixed point domains
2
partitioned abstract actor-oriented system model
code generator, parametrizable IP core
3.1
partitioned abstract actor-oriented system model
actor-oriented simulation + cycle-accurate simulator (JHDL, ModelSim, etc.)
isolated actors are mapped into cycle-accurate models
co-simulation
synthesis
cycle-accurate models are synthesized and mapped into FPGA-based prototyping platforms
3.2
partitioned abstract actor-oriented system model
actor-oriented simulation + FPGA prototyping
code generator, compiler
4
simulation management
actor-oriented simulation + cycle-accurate simulator and/or FPGA prototyping
reminder of the system model is mapped into software and execution platform
5
processor + memory + interconnect

Figure 1: Proposed Design Flow

4 Case studies

The design flow proposed in Section 3 is still an idealized view of how the design methodology should look like, and it still depends on further research and fine-tuning to be considered to be really applicable to industrial applications. To validate some of the steps of the flow and to identify and better understand some of the missing links of the chain, two case studies were performed and will be briefly described in the following subsections. The first one applies the proposed flow (until step 3.1) to the design of a wireless communication system. The second case study experiments with the co-execution of actor-oriented models and FPGA-based prototyping platforms in a “hardware-in-the-loop” approach (step 3.2 of the flow).

2.1 Design of a self-adaptable WCDMA receiver

This case study applies the concepts of actor orientation on the design of a self-adaptable receiver for WCDMA downlink UMTS standard. The goal is to take advantage of the possibility of using multiple models of computation to describe both the system and its usage scenario, aiming to validate the performance of the system under realistic conditions.

CDMA communication systems rely on spread spectrum modulation, which means that all users share the same channel. In the UMTS standard, each user is designed a unique code sequence and the spread is carried out by direct multiplication of the code by the user data. At the receiver side, knowing the code sequence of the user, the received signal can be decoded after reception, recovering the original data. However, in a multipath propagation environment such as a metropolitan area, the receiver receives several delayed copies of the same signal, making it difficult to recover the original signal. The receiver designed in this case study can self-adapt to the conditions of the transmission channel, combining the conventional rake receiver with maximal ratio combining (MRC) and a linear minimum squared error (LMMSE) equalizer.

The first step of this case study explores the actor-oriented modeling capabilities of Ptolemy II by creating models of both the receiver its usage scenario including user mobility patterns and transmission channel conditions (Figure 2). The accurate description of the application scenario increases significantly the complexity of the model, therefore it is implemented in a high level of abstraction using floating point arithmetic and relying mostly on pre-defined actors from the Ptolemy II library and a few custom actors coded in Java. Once the communication system is properly modeled, it can be simulated and designers can navigate through the design space by parametrizing the receiver and analyzing the bit error rate (BER) under different channel models. The implemented case study used the channel described in [HP02] over several simulation runs, varying (1) the number of users in the channel to validate the multi-user detection efficiency of the implemented receiver; and (2) the user mobility speed to evaluate the receiver capabilities to overcome Doppler effects.

After the analysis and profiling of the system model, the next step is to isolate the subsystems that are targeted to hardware implementation. This is done by placing conversion actors at the boundary of those subsystems. Such actors will convert the double precision floating point into fixed-point, so the quantization alternatives can be evaluated. The hardware implementation was divided in three subsystems: rake receiver, MRC and the MMSE linear equalizer. JHDL [BH98] was used to create cycle-accurate models of the subsystems that should be implemented in hardware. In order to validate the model transformation (manually done in this case study), every module written in JHDL should be encapsulated within an actor, integrated into Ptolemy II and co-simulated together with the whole model, allowing designers to evaluate the performance of the implemented subsystem under the constraints and conditions modeled within the testbench. Once the co-simulation results were satisfactory, JHDL netlist generation libraries are used and the generated netlist can be mapped to an FPGA prototyping platform. The results, detailed in [IPG05], have shown that the designed system presented smaller bit-error-rate than regular correlator-based receivers, and that this improvements are particularly significant in channels with increased Doppler effects, multi-path and multi-user interference. The synthesis of the cycle-accurate model allowed the evaluation of performance and power consumption of the designed system, which could run at a maximum frequency of 15.672 MHz (more than sufficient to handle the data rate of the UMTS standard) and would consume 115 mW (both estimations for an implementation using a Xilinx Virtex XCV800 device).

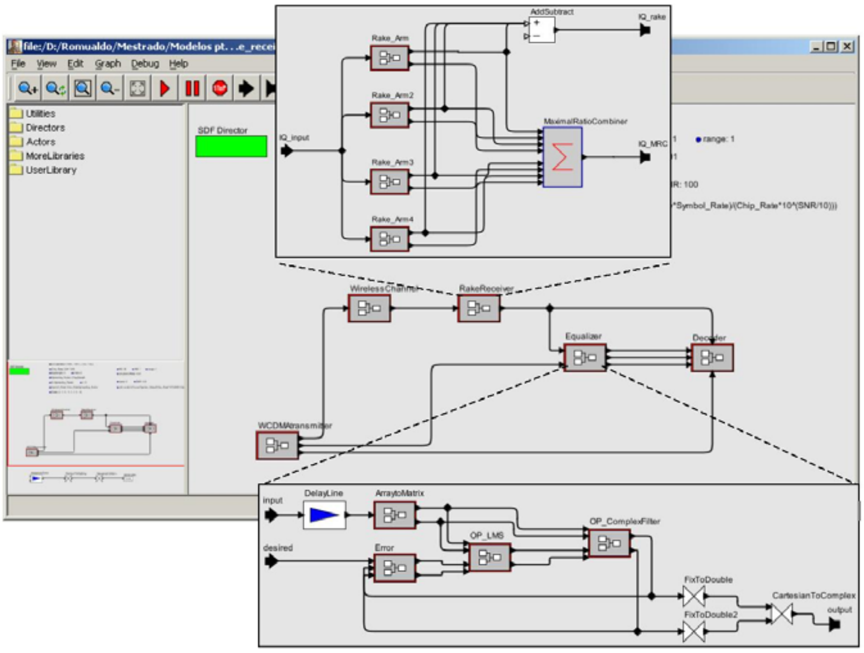


Figure 2: Top-level system view of WCDMA receiver in Ptolemy II

2.2 FPGA-based prototyping in actor-oriented environments

The integration of FPGA-based prototyping platforms and actor-oriented simulation environment allows the successive refinement of actors to the point where they can be implemented in hardware. In such scenario, a system may be initially designed as a set of actors described at high level of abstraction – usually the system itself and its usage scenario, as shown in the previous subsection – and parts of it are refined into models which can be synthesized and implemented in an FPGA. Then, those actors can be substituted by their respective FPGA implementations, which are encapsulated within actors and are integrated in the simulation environment. The final goal is to validate the design which was implemented in the FPGA with the original testbenches which were used in the actor-oriented model, so that it is possible to evaluate the performance of the FPGA implementation in comparison with the system-level model.

One important requirement is to allow the FPGA boards to be transparently accessible. Transparency is achieved when the hardware is abstracted as a software object, whereby its main features are offered through function/method calls. For this approach, a backend object, running in a host computer connected to the FPGA board, does the mapping between the hardware features and specific functionality [In03]. The backend's public methods allow for configuration of the board, information uploading to its RAM, execution triggering and information downloading from the RAM. Jini technology offers an infrastructure able to handle the communication between services and clients using proxies [Ar99], so the networking issues are abstracted away [In03].

This case study also uses Ptolemy II as the actor-oriented simulation environment. The selection, interconnection and execution of actors are accomplished graphically using Ptolemy II's GUI (Vergil) that allows for the construction of simulations. To integrate the FPGA proxies with Ptolemy II, the encapsulation of the Jini Client into a new actor is necessary. As a result, the backend capabilities are remotely added to the framework and made available to the user through an actor that can be selected, parameterized and interconnected with other actors. Once one Ptolemy II actor encapsulates the Jini client, remote boards can be connected, configured and used within the simulation of a system. The developed example depicted in Figure 3 uses a hardware implementation of the FFT algorithm, together with existing Ptolemy II Actors that are set to capture audio and display results. The remote FPGA implements the FFT algorithm, defined at the beginning of the simulation by the encapsulated client. During the simulation, local actors capture audio, sample it and deliver it to the remote actor. The remote actor packages a certain amount of samples and calls the corresponding method – using the proxy – to place the information in the RAM of the FPGA prototyping platform, wait for processing, and read results. The results come as the output of this method call. Finally, the actor packages the results as a token and sends it to the next local actor in the simulation, showing the results to the user. Further details of this case study can be found in [OIG05].

5 Conclusions

This paper presented an approach for a model-driven design flow for Systems-on-Chip using actor-orientation as the underlying framework for modeling and simulation. The major advantages of the multi-MoC approach supported by actor-orientation include (1) the co-simulation of models described in different levels of abstraction, (2) the use of complex testbenches, (3) the use of multiple granularities for time and (4) the possibility of visually compose systems.

The case studies have shown that actor-oriented SoC design is feasible and that despite of the high level of abstraction of the system models there are efficient mechanisms to support the customization and optimization of implementation-related features. Unlike similar approaches, this work is based on successive model refinement across abstraction layers, taking advantage of the possibility of co-simulation using multiple MoCs. Future research should focus on actor synthesis and code generation techniques, specially for the software part which was not covered by the presented case studies.

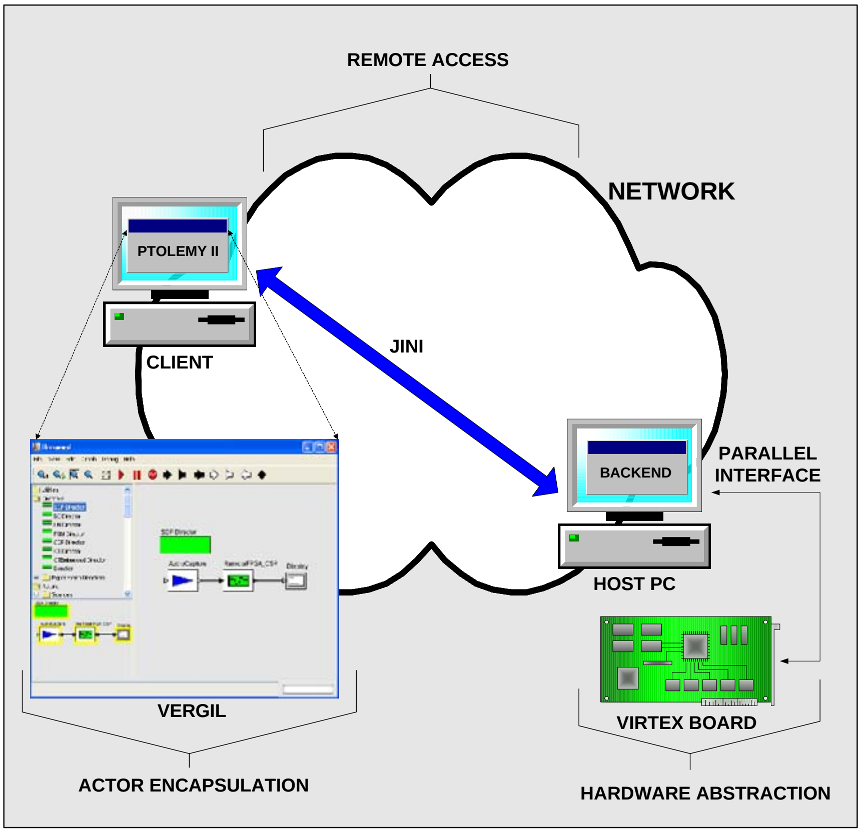


Figure 3: FPGA-based prototyping in actor-oriented environment

Acknowledgements

The authors would like to acknowledge Romualdo Begale Prudêncio and Diego Fernando Jiménez Oróstegui for their valuable contribution to the design and implementation of the case studies reported on this paper.

References

[Ar99] Arnold, K. et al.: The Jini specification. Addison-Wesley, 1999.

[BH98] Bellows, P.; Hutchings, B. L.: JHDL - An HDL for Reconfigurable Systems. In: Proceedings of FCCM '98, Napa Valley, 1998.

[HP02] Harada, H.; Prasad, R.: Simulation and software radio for mobile communication. Artech House Publishers, 2002.

[In03] Indrusiak, L. S. et al.: Ubiquitous Access to Reconfigurable Hardware: Application Scenarios and Implementation Issues. In: Design Automation and Test in Europe - Proceedings, München, 2003. IEEE Computer Society Press, 2003. p. 940-945.

[IPG05] Indrusiak, L. S.; Prudencio, R. B.; Glesner, M.: Modeling and Prototyping of Communication Systems Using Java: A Case Study. In: IEEE International Workshop on Rapid System Prototyping – Proceedings, Montreal 2005. IEEE Computer Society Press, 2005. p. 225-231.

[Le04] Lee, E. A. et al. : Heterogeneous Concurrent Modeling and Design in Java. Technical Memorandum UCB/ERL M04/27. UC Berkeley, 2004.

[LN03] Lee, E.A.; Neuendorffer, S.; Wirthlin, M.J.: Actor-Oriented Design of Embedded Hardware and Software Systems. Journal of Circuits, Systems, and Computers, v. 12, n. 3, p. 231-260, 2003.

[OIG05] Oróstegui, D. F. J.; Indrusiak, L. S.; Glesner, M.: Proxy-Based Integration of Reconfigurable Hardware Within Simulation Environments. In: Microelectronic Systems Education - Proceedings, Anaheim, 2005. IEEE Computer Society Press, 2005. p. 59 – 60.

[PS04] Patel, H. D.; Shukla, S. K.: SystemC Kernel Extensions for Heterogeneous System Modeling: A Framework for Multi-MoC Modeling and Simulation, Kluwer Academic Publishers, Boston, 2004.

Using an UML profile for timing analysis with the IF validation tool-set*

Iulian Ober University of Toulouse-II ober@univ-tlse2.fr
Susanne Graf VERIMAG, Grenoble susanne.graf@imag.fr
Yuri Yushtein NLR, currently at CIMSOLUTIONS B.V, NL yushtein@xs4all.nl

Abstract: This paper shows on hand of a case study the usefulness of the UML profile with real-time defined in the Omega project and of the IF validation tool-set. The case study is about intricate timing aspects arising in a small but complex component of the airborne Medium Altitude Reconnaissance System produced by NLR[1].

The purpose is to show how automata-based timing analysis and verification tools can be used by field engineers for solving isolated hard points in a complex real-time design, even if the press-button verification of entire systems remains a remote goal.

We claim that the accessibility of such tools is largely improved by the use of a UML profile with intuitive features for modeling timing and related properties.

1 Introduction

Designing a real-time system, so as to satisfy all its real-time properties, leads often to complex verification problems. This complexity is due to the fact that time is intrinsically a *global* notion, implicitly aggregating the *relative* and *local* timing conditions appearing in system design.

For defining non-trivial systems, it is nevertheless mandatory to conceive them in terms of local hypotheses and solutions. Consequently, in a component-based approach, designers seem to be condemned to build systems by component aggregation, without knowing a priori what effect this aggregation will have on the timeliness of each component and of the system as a whole. Some relevant examples of unexpected timing conditions resulting from this aggregation will be shown on the case study presented in this paper.

A solution to this problem consists in using automated tools to analyze the timeliness of a subsystem. There are two large classes of methods: model-checking which analyzes a semantic model algorithmically, and theorem proving. This paper is mainly about using a model-checking approach, which is more automatic. Very high-level parametric models are sometimes tackled better by proof-based techniques; but in general these models are elaborated by the verification experts rather than the engineers and the distance between these high-level models and the developed system may be quite important. E.g. when using duration calculus [CHR92] as verification framework, almost everything of the functional model has to be abstracted.

Model-checking techniques can be more easily applied to models that are obtained by starting from a functional model of a system (which is a design artifact normally available for any system), and which can be enriched by adding timing relevant information. Such a model can be quite naturally obtained by the designer and provides a faithful representation of the system under development. It can be directly analyzed with an appropriate simulation tool.

Nevertheless, automated verification tools have well-known limitations, and a first obstacle for putting these tools effectively to work, is that the designers have to understand them and build the models having these limitations in mind. From our experience, interesting insights in the timing aspects of a system are usually gained only when the (unrelated) details of the functional part are abstracted away. This means that a model must be decomposed into small functionalities, which makes property depending abstractions more easy to construct and even to mechanize. In our experience, not many engineers do this naturally, but they can easily learn it when they see the benefit.

*This work has been partially funded by the European OMEGA project (IST-2001-33522).

[1] National Aerospace Laboratory, The Netherlands.

The second obstacle is the complexity of the formalism for capturing a timing model and its properties. A good formalism is one that is intuitive for the designers and based on concepts they are already using. In the literature there are various extensions of temporal logics with quantitative time operators, which have the required expressiveness. However, from our experience, property formalisms based on familiar concepts (like state machines) are more easily accepted by the users and are more expressive.

In this paper, we present the results of a case study conducted jointly by experts and industrial users, in which meaningful results about timing were obtained by analyzing a custom made model using a user friendly UML-based validation tool. The rest of the paper is structured as follows: §2 presents the case study, with focus on the timing aspects. §3 presents the approach and the model obtained for this case study using a specific formalism (the OMEGA UML profile), the main results of timing validation and the techniques employed in this experience. In §4 we discuss some conclusions that might be drawn from this study.

2 The MARS system

The acronym MARS stands for Medium Altitude Reconnaissance System. It controls a high resolution photo camera embedded in a military aircraft, taking pictures of the ground from medium altitude. The system counteracts the image quality degradation caused by the forward motion of the aircraft by creating a compensating motion of the film during the film exposure. The system is also responsible for annotating the frames with the current time and position. The system also performs health monitoring and alarm processing functions.

Exposure control (Forward Motion Compensation and Frame Rate) as well as annotations are being computed in real-time based on the current aircraft altitude, ground speed, navigation data (latitude, longitude, heading), time-of-day, etc. These parameters are acquired from the avionics data bus of the aircraft.

2.1 The Databus Manager

For the purpose of this case study we concentrated on a sub-system which presents interesting timing problems, called Databus Manager (DM); it monitors the health of the data bus controller and, in general, of any communication going trough the data bus.

The system receives *data* concerning altitude and navigation from other components of the avionic system. The DM component supervises the reception of data messages and provides a *status* used by the system's alarm logic. In addition, the DM periodically polls the databus controller status and changes its own status accordingly to *Operational*, *BusError* or *ControllerError*. The precise requirements on the DM status computation are described below.

The two types of *data* inputs of the DM are received periodically, with some period ($P = 25ms$ in the concrete example) and jitter ($\pm J = 5ms$), and may occasionally get lost. The periods are not synchronized and may have any offset (smaller than the period). Figure 1 shows a possible configuration of the reception windows along the time axis (windows in which no message reaches the DM are marked with KO). The basic functional requirements on the DM status are:

- Controller failure leads to a change of status to *ControllerError*. Recovery leads to *BusError*.
- Status changes from *BusError* to *Operational* when two consecutive messages are (correctly) received from both sources (assuming no controller error).
- Status changes from *Operational* to *BusError* when three consecutive messages from a source are lost.

Note that these requirements do not define *when* the status change must take place. In fact, maximal reactivity is desirable. Two reactivity measures (at least) can be defined:

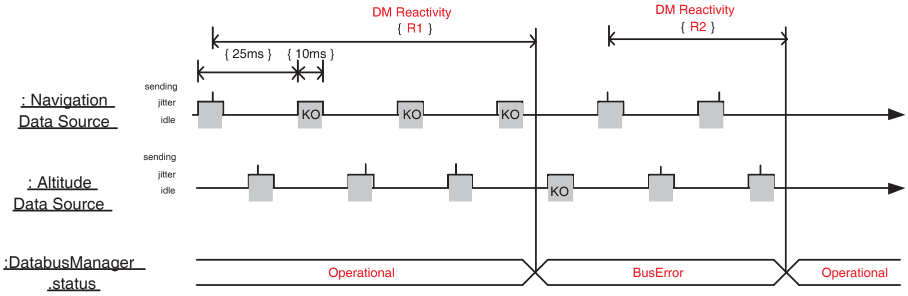


Figure 1: Timing of the message reception windows, DM status and reactivity measures.

- reactivity to losses, defined as the upper bound that DM guarantees for the time ($R1$) between the last correctly received message from the source causing a switch to *BusError*, and the actual moment of the switch.

  Simple problem analysis shows that the optimal $R1$ is $85ms$ ($3P + 2J$), in the case of fixed communication time between sender and receiver. This is the ideal reactivity that the DM design should try to approach.

- reactivity to recovery, defined as the upper bound that DM guarantees for the time ($R2$) between the first message in a series of correct messages leading to a switch to *Operational*, and the actual moment of the switch.

  In this case, the optimal $R2$ depends on the offset between the periods of the two data sources. However, even in the worst case, it is less than $60ms$ ($2P + 2J$).

Our experiments are described in the next section. They had two goals: (1) to check that the proposed designs verify the above mentioned functional properties, and (2) to determine the reactivity bounds offered by the different proposed designs (and point out the optimal solution).

3 UML modeling and validation experiments

3.1 Background on the OMEGA profile and the IFx toolset

The MARS sub-system was modeled using the OMEGA UML profile and timing and functional validation was performed using the IFx toolset. Here, we briefly introduce these technologies.

The OMEGA profile defines an operational semantics for a subset of UML, designed to suit the needs of designers of real-time embedded systems. On the *functional* side, the semantics defines aspects pertaining to control (like the rules governing concurrency) and communication primitives. These aspects are handled similarly as in the profile of the Rhapsody tool, and they are detailed in [DJPV03, DJPV05].

The *timing* aspects are described in detail in [GOO05]. The profile is compatible with the basic time related notions of UML 2.0 by defining a series of lightweight extensions to UML for describing time-driven behavior using timers, clocks, and timed guards. In addition, it allows to define transition urgency for categories of transitions[2]. For the expression of timing and functional requirements, the OMEGA UML proposes to use a notion of *events* and *observer objects*. Events are any semantic level state changes, similar as the notion of *timed event* in the SPT profile [OMG02]; in order to make this notion concrete, a notation has been defined for being able to name all semantic events by referring to a syntactic entity. Observers are characterized by a state machine which reacts to (semantic level) events and conditions occurring in the

[2] this concept is taken from timed automata with urgency [BS97]

system, and which acts as an acceptor of system executions – by use of states stereotyped with <<`error`> > as final states. An example of a property expressed by an observer can be seen in figure 5.

IFx [OGO05] is a toolset providing extended simulation and verification functionalities for OMEGA UML models. The core of the tool is a state space exploration engine for systems consisting of extended communicating timed automata (IF [BGM02, BGO$^+$04]). In order to scale to complex models, IF provides several optimizations and and supports abstraction. The tool implements static and dynamic optimizations like dead variable factorization, dead code elimination, partial-order reduction and abstract interpretation of clocks. All optimizations strongly preserve timed safety properties which are of interest in the MARS system. In addition, the tool supports simple abstractions which preserve satisfaction of safety properties, but may show spurious counter-examples.

3.2 Overview on the UML model for MARS

The architecture of the MARS model as proposed by the designer of the system, is shown in the UML context diagram in Figure 2[3]. The main component is the *DatabusManager* object which maintains the global status and monitors message loss. For simplicity, the designer has separated the polling of the bus controller in a different object, the *ControllerMonitor*.

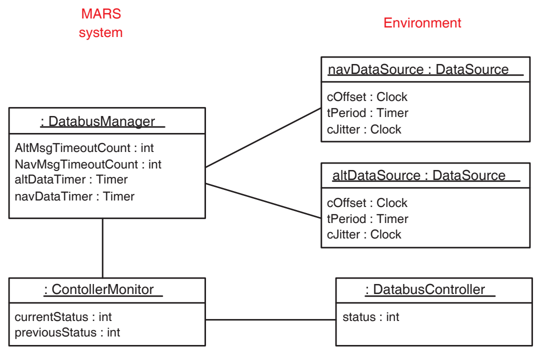


Figure 2: Structure of the MARS model.

In order to verify the DM under the assumptions on message arrival and controller errors mentioned in §2.1, the OMEGA profile allows to model the environment using the same concepts as for modeling the system, in particular an explicit object with the behavior expressed by the assumption. In Figure 2, we see therefore three environment objects corresponding to the altitude data source, the navigation data source and the bus controller.

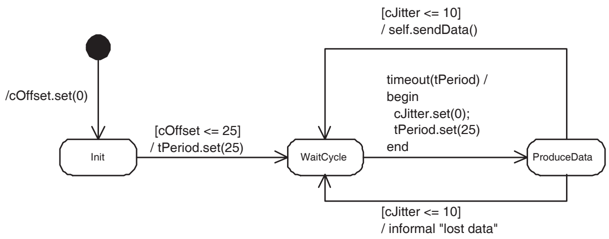


Figure 3: Environment model : state machine of the data sources.

In particular for modeling the environment, the possibility to express nondeterministic behavior is important. This is allowed in the Omega profile. For example, Figure 3 shows the state machine of data sources,

[3] associations express here the fact that the corresponding objects may communicate through signals or method calls

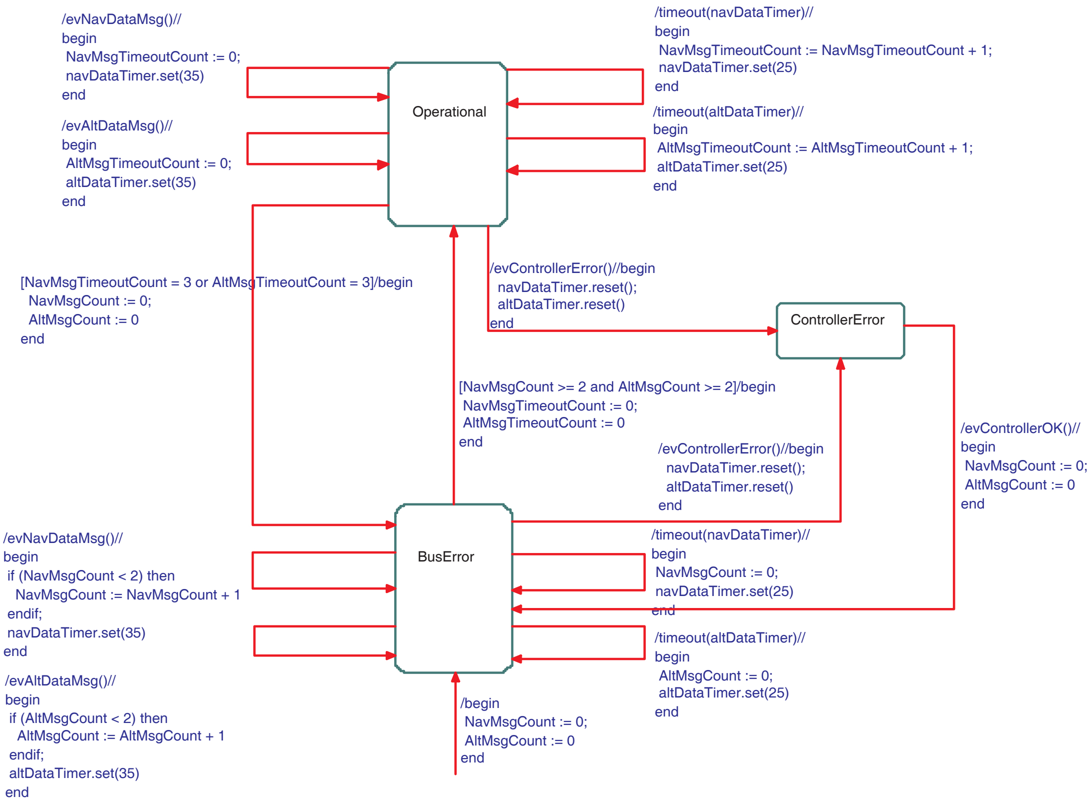


Figure 4: State machine of the DatabusManager.

using interval conditions on clocks to model the nondeterminism introduced by the starting time and by jitter. This state-machine indeed describes a data source with the required period and jitter as all transitions of environment objects are interpreted as *delayable*[4], that is, once they are enabled, they will be taken before their time guard becomes false or they may be disabled by some discrete transition. Moreover, Zeno computations[5] are not valid computations which guarantees in this example that the computation cannot "get stuck" in any state.

As we will see later, the requirements concerning the DM can be achieved in several ways. The first design provided by the engineer, shown in Figure 4, consisted of a single state machine with transitions triggered by events from both data sources ($evAltDataMsg$, $evNavDataMsg$) and from the *ControllerMonitor* ($evControllerError$, $evControllerOK$), or by timeouts corresponding to message loss detection ($altDataTimer$, $navDataTimer$). The principle is to use a timer to measure the duration of the period for each Data source — where the end of the period is defined by the reception of a message, where the timer is rearmed, or a timeout — and to always keep track of the number of consecutive received, respectively lost, messages. Notice that in this design we have chosen system transitions to be *eager*, that is they do occur at the earliest point of time at which they are enabled[6]. must progress,

[4] according to the terminology defined in timed automata with urgency [BS97]

[5] infinite computations with finite time progress

[6] Note also, that transitions are event triggered rather than time triggered as *timeout* is the event associated with a time condition. This is equivalent, but closer to operational way of thinking of the designers

3.3 Expressing properties and first evaluation results

Both, the functional and the reactivity properties described in § 2.1 can be expressed as observers to be verified on this model. Figure 5 shows the observer checking a bound guaranteed for $R1$, that is the maximal delay needed for transmission problem detection (see section 2.1). Note that observer transitions *synchronize* with observed events, and thus take place at the same time point as the observed event. Note also that this observer monitors only one data source (the altitude data source); we argue that the failure of the other source can only bring the DM into the $BusError$ status earlier, thus the maximum value for $R1$ is exposed when the other source does not fail (or it's failure is not observed).

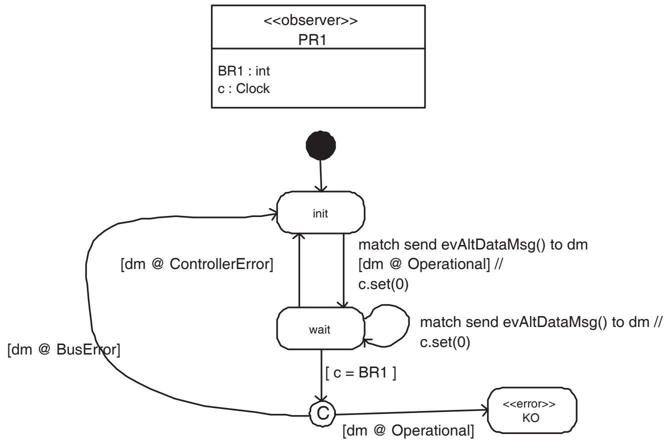


Figure 5: Observer for verifying the reactivity bound for $R1$.

All functional and reactivity properties were verified against the initial design presented above.

Due to state explosion problems encountered, a simplifying assumption was made first on the environment: the cycles of the two data sources are synchronized (i.e., their periods start always at the same time; their data may nevertheless be sent at different moments due to jitter). This assumption is not conservative[7] as the reaction time to message loss may (and turn out to) be longer when the two sources are not synchronized. In order to fully verify the properties, a different model, which is a conservative abstraction, is presented later on.

Nevertheless, this initial model was useful for understanding the potential problems, for debugging the model and ruling out some variants that had been proposed to increase efficiency. Under the simplifying assumption, all functional properties have finally been proved to hold on debugged versions of the DM design. An interesting outcome is that very similar designs may present different reactivity bounds.

For example, consider a slight variant of the design model presented above, in which in the *operational* status a *long timeout* is used, detecting the absence of messages during three consecutive periods, instead of detecting absence of individual messages and counting them. At a first sight, this new version looks more efficient as it does not need counter when the status is *operational*.

Using different variants of the reactivity constraints defined by the observer in Figure 5, we have determined with the help of our verification tool that the initial design has a better reactivity (85ms[8]) than the new one (only 110ms). As the motivation for the entire case study was to gain reactivity with respect to the existing synchronous design, which observes events only at fixed time points, this is not an acceptable solution.

The diagnostic traces provided by the model checker show that the difference stems from the way timers are handled at the transition from the *BusError* to the *Operational* status: in the initial version, timers are not affected by a status change, they just count periods of the data sources, while in the new version, the

[7] it leads to a simplified model that allows to find bugs, but we are not allowed to deduce correctness of the initial model from the fact that this model does satisfy some property

[8] that means it achieves the optimal reactivity as explained in section 2.1

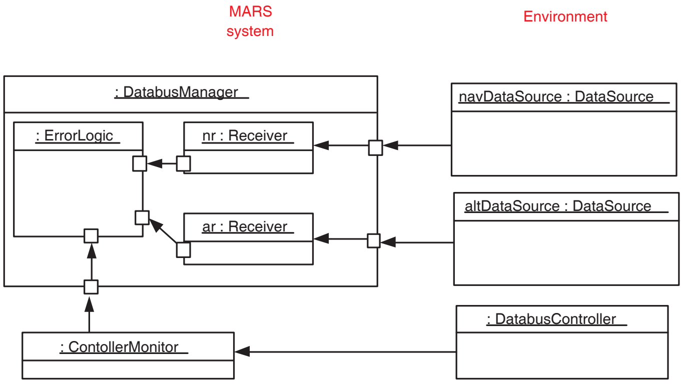


Figure 6: Decomposition of the DM.

long timer is not needed in *BusError* status and initialized when entering the operational mode. This might delay the detection of a bus error occurring right at this moment by an entire period. In order to correct the problem, the timer must be set depending on the actual "age" of the "period timer" which is used when the status is *BusError*.

This kind of errors is quite common when trying to optimize a design, and the tools were very helpful by checking after any modification all the properties. This allows to get immediate feedback as running the verification is generally really fast (see also the table at the end of the section). Another group has used in parallel a tool based automatic or interactive theorem proving. They could prove at the end a parametric version of the property which is impossible to prove with a model-checker like ours, but they were extremely thankful to the extremely quick feedback provided by our tool: once we had a (simplified) working model on which the properties can be shown to hold, one can analyze small modifications of the UML model, just by "pushing the button" to translate the model and then run the verification of all properties, or "find a bug".

3.4 Use of a compositional model and abstractions

In order to fully verify the desired properties without making the (unrealistic) assumption that both data sources are synchronized, we have proposed to use a model of the DM that is itself a composition of smaller entities (see Figure 6), in particular

- A *Receiver* component for each data source; it supervises the messages of a single source and keeps track of the message status in the last 3 windows and sends this status by means of *evCnt* messages at the end of each period.
- An *ErrorLogic* component which, based on the *evCnt* messages from the different *Receivers*, defines the status. The status changes from *BusError* to *Operational* when all *Receivers* have received correct messages during the last 2 windows, and from *Operational* to *BusError* when at least one *Receiver* has not received any correct message during the last 3 windows.

Using this model, we could verify all properties — or rather adaptations of them — using a compositional and conservative abstraction.

The abstraction used consists in replacing one *Receiver* with a chaotic abstraction *ReceiverAbs* which may send *evCnt* with any parameter at any time. This is a very rough over-approximation of the source–receiver pair, but it proved to be sufficient for preserving the desired properties. The abstraction is particularly interesting as it represents an over approximation of an arbitrary number of data receivers, meaning that it allows to verify the DM for the case with an arbitrary number of such data sources[9].

[9] always under the hypothesis that the time for taking decisions remains "negligible"

Configuration	Number of states	Number of transitions	User time
Initial model with only one source (no *CM* polling) (*non-conservative*)	1084	1420	$< 1s$
Initial model with two synchronized sources (no *CM* polling, *non-conservative*)	99355	151926	$36s$
Initial model with two de-synchronized sources (no *CM* polling) (*conservative* – exploration doesn't terminate)	> 1136768	> 1676126	$> 9m30s$
Abstract model, 10ms *CM* polling (*conservative* – does not terminate)	> 1494864	> 701120	$> 8m12$
Abstract model (no *CM* polling) (*non-conservative*)	118690	174871	$45ms$
Abstract model with non-det. *CM* polling (*conservative*)	155166	263368	$1m21s$

Figure 7: Verification times and state spaces for different verification configurations.

A second conservative abstraction used consists in replacing the deterministic polling cycle of the *ControllerMonitor* (10ms in the initial model) by a completely non-deterministic polling policy. While this introduces new executions, impossible in the initial model, the resulting state space is smaller as many previously disjoint states are grouped together[10].

The table in Figure 7 below shows the size of the state space and the processing time for several configurations of the MARS system which allows to draw some conclusions on the efficiency of the use of compositional models and in particular compositional abstractions. In particular, desynchronizing two resources has a tremendous effect on the size of the state space which is in fact due to the simultaneous presence of jitter and desynchronization. Notice that the effect is much more important than it looks like as we have stopped the exploration when, after reaching 1 mio states and 10 minutes, the state space was still growing rapidly — experience told us that it was likely not to be worth to wait until reaching 10 mio states; even if it would converge by then, what we didn't anticipate really, the result was not very useful for us, as we wanted to be able to rerun experiences on variants in short time. As we have not run our experiences on particularly well equipped machine (especially in memory), this means that we can still gain a few orders of magnitude and handle slightly more complex systems.

The use of both types of conservative abstractions leads to a state space of about the same size as the much simpler system with synchronized sources, which is still precise enough to satisfy all properties.

4 Conclusion

By using a case study as support, we have shown both, the convenience of the OMEGA UML profile for the expression of timed models and timed properties and the usefulness of the IF front-end for UML which allows for both flexible interactive simulation and complete state space exploration for debugging and verification of UML models.

We believe that the experiment presented shows that timing analysis tools can be used efficiently for solving isolated, hard timing problems in a UML design, even if fully automated verification for large designs remains a remote goal. Also we believe that more systematic use of functional decomposition as used in the example, can definitively help to make possible the verification of much larger designs in a compositional fashion, as there is no need for the verification of a model in which all parts are described in all details.

[10] but only if a symbolic representation of time constraints is used

The use of the OMEGA UML profile to capture timing properties has favored a very quick learning and adoption of our tools by experienced UML designers. Without the knowledge of a verification expert, the designers were able to use even advanced techniques like abstractions.

The relaxation of timing constraints — such as the abstraction from the polling period in the example — shows to be a very efficient abstraction technique in such models, and it is usually very simple to model. This kind of abstractions is always conservative for the satisfaction of (timed) safety properties. On the other hand, can introduce spurious error traces. However, in the MARS example this has never occurred, showing first, that with some exercise, a designer can learn to use abstractions which do not break the verified properties. And second, that the designers tend in the first place to build over constrained models. The reason is probably that they are strongly influenced by the requirement that programs must be deterministic, and they apply this also to specifications even if this is not needed for satisfying the requirements.

We have also found out during the experiments that some methodological guidelines for writing observers and for using the IFx toolbox are necessary during the learning process. A set of guidelines has been developed as a side result of this teamwork (see also [OGL05]).

References

[BGM02] Marius Bozga, Susanne Graf, and L. Mounier. IF-2.0: A Validation Environment for Component-Based Real-Time Systems. In *Proceedings of Conference on Computer Aided Verification, CAV'02, Copenhagen*, number 2404 in LNCS. Springer Verlag, June 2002.

[BGO$^+$04] Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis. The IF toolset. In *SFM-04:RT 4th Int. School on Formal Methods for the Design of Computer, Communication and Software Systems: Real Time*, number 3185 in LNCS, June 2004.

[BS97] S. Bornot and J. Sifakis. Relating Time Progress and Deadlines in Hybrid Systems. In *International Workshop, HART'97, Grenoble*, LNCS 1201, pages 286–300. Spinger Verlag, March 1997.

[CHR92] Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A Calculus of Durations. *Information Processing Letters*, 40(5):269–276, 1992.

[DJPV03] Werner Damm, Bernhard Josko, Amir Pnueli, and Angelika Votintseva. Understanding UML: A Formal Semantics of Concurrency and Communication in Real-Time UML. In Frank de Boer, Marcello Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Proceedings of the 1st Symposium on Formal Methods for Components and Objects (FMCO 2002)*, volume 2852 of *LNCS Tutorials*, pages 70–98, 2003.

[DJPV05] Werner Damm, Bernhard Josko, Amir Pnueli, and Angelika Votintseva. A discrete-time UML semantics for concurrency and communication in safety-critical applications. *Science of Computer Programming*, 2005. (to appear).

[GOO05] Susanne Graf, Ileana Ober, and Iulian Ober. Timed annotations in UML. *STTT, Int. Journal on Software Tools for Technology Transfer*, 2005. under press.

[OGL05] Iulian Ober, Susanne Graf, and David Lessens. A case study in UML model-based dynamic validation: the Ariane-5 launcher software. submitted, 2005.

[OGO05] Iulian Ober, Susanne Graf, and Ileana Ober. Validating timed UML models by simulation and verification. *STTT, Int. Journal on Software Tools for Technology Transfer, 2004*, 2005. Under press.

[OMG02] OMG. Response to the OMG RFP For Schedulability, Performance and Time, v. 2.0. OMG document ad/2002-03-04, March 2002.

Zur Kosteneffektivität des modellbasierten Testens

Alexander Pretschner

Informationssicherheit, ETH Zürich

Abstract: Annahmen zum kosteneffektiven Einsatz des modellbasierten Testens werden aufgezählt. Existierende empirische Evidenz für das Zutreffen dieser Annahmen wird genannt und die Notwendigkeit weiterer empirischer Studien betont.

1 Einleitung

Testen umfasst eine Menge von Aktivitäten, die Differenzen zwischen Ist- und Sollverhalten eines Systems aufdecken oder das Vertrauen in die Abwesenheit solcher Differenzen erhöhen sollen. Das Verständnis von Spezifikationsdokumenten, gewissermaßen ein mentales Modell des zu testenden Systems (system under test, SUT) und seiner Umwelt, erlaubt einem Testingenieur die Definition von Testfällen (im folgenden kurz „Tests“): Ein- und erwartete Ausgaben, die mit den tatsächlichen Ausgaben des SUT verglichen werden. Der Aufwand des Testens wird übereinstimmend mit 50±20% der gesamten Entwicklungskosten angegeben, wobei diese Zahlen bisweilen die Identifikation des Problems (in Code oder Spezifikation) und auch das Debugging beinhalten [14].

Kernidee des modellbasierten Testens (MBT) ist, das mentale Modell durch explizite Verhaltensmodelle von SUT bzw. dessen Umgebung zu ersetzen und so insbesondere den Prozess der Definition von Tests explizit, nachvollziehbar, reproduzierbar und letztlich unabhängig von den speziellen Fähigkeiten eines einzelnen Testingenieurs zu gestalten. Das Verhalten des Modells des SUT wird als Sollverhalten für das SUT interpretiert, geeignet auszuwählende Abläufe des Modells des SUT dementsprechend als Tests. Wenn das Modell des SUT nicht genügend Informationen über die erwarteten Ausgaben codiert und dementsprechend hauptsächlich ein Umgebungsmodell vorliegt, kann daraus nur der Eingabeteil der Testfälle abgeleitet werden, und die erwarteten Ausgaben müssen vom Tester zur Verfügung gestellt werden.

Kern dieses Aufsatzes ist die explizite Benennung von Annahmen über den kosteneffektiven Einsatz des MBT, die häufig implizit getroffen werden. Angesichts ganz erstaunlicher Fortschritte in technologischer Hinsicht wird – hier bewusst nicht abschließend – analysiert, unter welchen Bedingungen MBT kosteneffektiv eingesetzt werden kann und unter welchen Bedingungen der Einsatz fragwürdig erscheint. Das übergeordnete langfristige Ziel ist dabei die Identifikation entsprechender organisatorischer und fachlicher Erfolgsfaktoren. Zweck dieses Aufsatzes ist das Schaffen einer Diskussionsgrundlage für den Workshop.

Beitrag. Dem Verfasser sind keine Arbeiten anderer Autoren zu prinzipiellen Überlegungen zum erfolgreichen Einsatz des modellbasierten Testens bekannt. Aufbauend auf früheren Papieren [22,27], reflektiert dieser Aufsatz die aktuellen Gedanken.

Abgrenzung. In diesem Papier wird auf den Test von Funktionalität fokussiert; Usability- und Stresstests und die Eigenheiten des Testens objektorientierter Software werden nicht betrachtet. Struktur- oder Nutzungsfallmodelle zur automatisierten Generierung von Testinfrastruktur werden ebenfalls ignoriert. „Modelle“ bezeichnen Artefakte, die eine Vereinfachung des Verhaltens eines SUT oder seiner Umwelt darstellen und die (insbesondere) zum Zweck des Testens erstellt wurden [26]. Es wird angenommen, dass Modelle so präzise bzw. formal sind, dass daraus zumindest im Prinzip Tests automatisch erzeugt werden können. Von konkreten Modellierungstechniken und –sprachen wird abstrahiert, und die offenkundigen Zusammenhänge mit dem modellbasierten Test von Hardware bzw. Chipdesigns werden nicht diskutiert.

Überblick. Abschnitt 2 präsentiert zentrale Ideen des MBT. Abschnitt 3 listet fundamentale Annahmen an den erfolgreichen Einsatz des MBT auf, die als Erfolgsfaktoren gelesen werden können. Sofern verfügbar, wird über publizierte Evidenz berichtet. Abschnitt 4 zieht Schlussfolgerungen und benennt attraktive Forschungsrichtungen.

2 Modellbasiertes Testen

Testen beinhaltet drei einander ggf. überlappende Phasen: die Auswahl, Durchführung und Auswertung von Tests. Für die Auswahl von Tests werden Modelle von SUT, Modelle von dessen Umgebung und Modelle von Fehlern verwendet. Die Anzahl möglicher Tests ist üblicherweise unendlich oder zumindest sehr groß. Der ökonomische Standpunkt führt zum Wunsch nach einer möglichst kleinen Anzahl von zu erzeugenden, durchzuführenden und auszuwertenden Tests. Aus der Perspektive der Qualitätssicherung muss diese Anzahl jedoch ausreichend hoch sein, um die Anzahl verbleibender Fehler auf ein akzeptables Minimum zu reduzieren. Testfallableitung, ob automatisch oder manuell, muss also das Problem lösen, im folgenden Sinn gute Tests zu generieren: Ihre Erzeugung soll billig sein und ihre Auswertung ebenfalls, insbesondere in Bezug auf die Zuordnung von Fehlern (failures) zu Fehlerursachen (faults, errors). Außerdem sollen sie mindestens alle „schweren“ und „häufig“ auftretenden Fehler finden. Schließlich ist der Aufwand der Testdurchführung zu minimieren, was bei gemischten HW/SW-Systemen nicht immer unproblematisch ist.

Selektionskriterien. Zunächst unabhängig von der Verwendung expliziter Modelle wählen Tester als Annäherung an diese Ziele Tests anhand funktionaler, struktureller, fehlerbasierter und stochastischer Kriterien aus. Alle diese Kriterien können auch zur Bemessung einer Testsuite und als Testendekriterium verwendet werden [29]. Sie müssen vom Benutzer gefordert oder definiert werden: MBT basiert immer auf mindestens zwei vom Menschen gelieferten Eingaben, dem Modell und dem Selektionskriterium.

Strukturelle Kriterien fordern, dass während der Ausführung eines Programms oder Modells bestimmte Substrukturen von Datentypen, Kontroll- und Datenflussgraphen abgedeckt werden. Neben dem unklaren Zusammenhang von Abdeckung auf (SUT- und Umgebungs-) Modell und Code sind diese Kriterien in ihrer Fähigkeit, Fehler aufzuspüren, nicht unumstritten, auch im Vergleich mit randomisierten Tests (Referenzen in [24]). Ein großer Vorteil struktureller Kriterien liegt in ihrer Quantifizierbarkeit und darin, dass sie zumindest prinzipiell die vollautomatische Ableitung von Tests erlauben. *Funktionale Kriterien* zielen darauf ab, Funktionalitäten eines SUT zu isolieren und entsprechende Tests zu definieren. Das geschieht ad hoc, in Form von Szenarien oder Constraints über Abläufen. Wenn explizite Modelle der Umwelt des SUT oder von Anforderungen, auch in Form von Eigenschaften, vorliegen, können strukturelle Selektionskriterien auch auf diese Modelle angewendet werden. Unterstützung bei der Definition konkreter funktionaler Kriterien scheint in methodischer, aber nicht in technologischer Hinsicht möglich. *Fehlerbasierte Kriterien* verwenden empirisches Wissen um vermutete Fehler für die Selektion. *Stochastische Kriterien* schließlich basieren auf Wahrscheinlichkeitsverteilungen von Eingabedaten. Eine Gleichverteilung führt zu rein randomisierter Erzeugung des Eingabeteils von Tests, während die Verwendung von Nutzerprofilen [19] bzw. stochastischer Umweltmodelle [28] besonders attraktiv erscheint, weil sie unter bestimmten Bedingungen die Anzahl der verbleibenden, im Feld auftretenden Fehler minimiert

Testfallgenerierung. Aus einem Modell und einem adäquat operationalisierten Selektionskriterium leitet ein menschlicher oder maschineller Testfallgenerator Abläufe des Modells ab. Wenn hierfür ausschließlich ein Umgebungsmodell verwendet wurde, müssen die erwarteten Ausgaben des SUT hinzugefügt werden Projektionen von Abläufen des Modells auf Ein- und Ausgabe sind dann Tests für das SUT. Automatisierte Testfallgenerierung basiert auf dedizierten Suchalgorithmen, symbolischer Ausführung, Model Checking, deduktivem Beweisen oder Erfüllbarkeitsüberprüfern für propositionale Logik.

Abstraktion. In methodischer Hinsicht ergibt das MBT nur dann Sinn, wenn das Modell des SUT abstrakter als das SUT ist, also einen echten Informationsverlust beinhaltet [23]. Andernfalls würde der Aufwand, das Modell zu validieren, genau dem Validierungsaufwand für das SUT entsprechen (bisweilen wird argumentiert, dass der Abstraktionsunterschied nicht methodisch notwendig ist, weil allein die Redundanz von Modell und SUT zum Aufdecken von Problemen führt.) Das impliziert einerseits, dass nur das im Modell codierte Verhalten getestet werden kann und andererseits, dass die verschiedenen Abstraktionsniveaus überbrückt werden müssen (verschiedenen Abstraktionen im modellbasierten Testen sind an anderer Stelle beschrieben [21]): Der Eingabeteil des Tests wird konkretisiert, bevor er auf das SUT angewendet wird, und die Ausgabe des SUT wird abstrahiert, bevor sie mit der erwarteten Ausgabe des Tests, also des Modells, verglichen wird. Im Extremfall kann das Abstraktionsniveau so hoch sein, dass nur zwischen „Ausnahme geworfen“ und „keine Ausnahme geworfen“ oder „funktioniert“ und „funktioniert nicht“ (s. etwa eine Studie im Bereich des evolutionären Testens [7]) unterschieden wird. Konkretisierung und Abstraktion werden von dedizierten Adapterkomponenten implementiert.

Redundanz. Schließlich basiert Testen immer auf einer Form von Redundanz. Mit der Ausnahme von Stress- und Leistungstests ist es fragwürdig, ein einzelnes Modell für die Generierung von Produktionscode und Tests zu verwenden: Das SUT würde gewissermaßen gegen sich selbst getestet. Auf diese Art können allerdings Umweltannahmen und Codegeneratoren getestet werden. Verschiedene Szenarien des MBT, die die zeitliche Abfolge von Modell- und Systementwicklung berücksichtigen, werden an anderer Stelle diskutiert [23].

3 Annahmen und Evidenz

Dieser Abschnitt, der Kern des vorliegenden Papiers, fasst die Annahmen für den erfolgreichen Einsatz des MBT zusammen. Die Diskussion ist dann aufgespalten in eine Diskussion über Annahmen, für die dem Verfasser keine explizite Evidenz bekannt ist und in die Annahmen, für die Evidenz publiziert ist.

3.1 Annahmen

Häufig getroffene Annahmen über den kosteneffektiven Einsatz des MBT beziehen sich auf die im Folgenden aufgeführten Aspekte. Die Annahmen beziehen sich auf den Test von Funktionalität des SUT; modellbasierte Tests für Codegeneratoren, Compiler und Umweltannahmen werden dabei explizit aus der Diskussion ausgenommen.

1. *Modelle, Anforderungen und Spezifikationen:* Der Vorgang des Modellierens an sich hilft beim Verständnis und bei der präzisen Formulierung der Anforderungen (1a). Wenn ein Entwicklungsprozess auf sehr präzise Spezifikationen angewiesen ist, wie das etwa im Zusammenspiel von OEMs und Zulieferern im automobilen Bereich der Fall ist, dann sind Modelle als Spezifikationen ohnehin notwendig. Die Möglichkeiten des MBT führen dann zu einem zusätzlichen Wert des Modells (1b).
2. *Existenz adäquater Modelle:* Es gibt ein Modell, bei dem der Tradeoff zwischen methodisch notwendiger Abstraktion und zum Testen erforderlicher Detaillierung gerechtfertigt werden kann (offenbar bezieht sich diese Annahme nur auf die Ausprägungen des MBT, bei denen das Ausgabeverhalten des SUT hinreichend detailliert codiert wird). Weil dieses Modell eine Vereinfachung darstellt, ist es leichter zu validieren als das entsprechende SUT.
3. *Effektivität:* Modellbasiertes Testen findet (potentielle) Fehler im SUT.
4. *Relativer Aufwand und relative Qualität:* Der Aufwand, Modelle und modellbasierte Testselektionskriterien zu erstellen, zu validieren und zu warten ist kleiner als der Aufwand, eine ohne Modelle erstellte Testsuite zu erstellen, zu validieren und zu warten. Das gilt insbesondere für Modifikationen des SUT etwa bei sich ändernden Anforderungen. Wenn der Aufwand nicht geringer ist, rechtfertigt doch zumindest die höhere „Qualität“ der Tests den erhöhten Ressourceneinsatz. Bisweilen wird angenommen, dass große Testsuiten gar nicht gewartet werden können, dass das für Modelle hingegen aber sehr wohl der Fall ist.

5. *Wiederverwendung:* Im Kontext von Produktlinien oder allgemein variantenreicher Systeme können Modelle, Testselektionskriterien und Adapterkomponenten leichter wiederverwendet werden als entsprechend manuell erstellte Testsuiten.
6. *Testanzahl:* Wenn ein Modell einmal erstellt ist, können bei minimalem Aufwand beliebig viele (und im Fall reaktiver Systeme auch beliebig lange) Tests erzeugt werden. Eine größere Anzahl an Testfällen ist i.a. einer kleineren vorzuziehen.

3.2 Diskussion der Annahmen ohne dem Verfasser bekannte publizierte Evidenz

Im nächsten Abschnitt 3.3 wird empirische Evidenz für die Annahmen (1), (3) und (4) angeführt. Im verbleibenden Teil dieses Abschnitts erfolgt zunächst eine kurze Diskussion der anderen Annahmen, für die dem Verfasser keine publizierte explizite Evidenz bekannt ist.

In Bezug auf die Existenz adäquater Modelle (Annahme 2) ist festzuhalten, dass bei der Verwendung von Modellen zur automatischen Ableitung von sowohl Code als auch Testfällen stets ein fundamentaler Widerspruch zwischen den Vorteilen einer abstrakten Beschreibung und der notwendigen Detailliertheit der generierten Artefakte besteht. Abstraktion tritt in mindestens zwei Ausprägungen auf, nämlich in der Form der Kapselung und in der Form expliziten Informationsverlusts, von denen im MBT insbesondere die zweite starke Relevanz besitzt.

Abstraktion durch Kapselung wird etwa durch dedizierte Programmiersprachenkonstrukte (etwa Konditionale, Schleifen, Unterprogramme, Garbage Collectors), Middleware, Betriebssystemschichten und Bibliotheken erzielt. Die nicht explizit aufgeführte Information wird durch das Laufzeitsystem oder den Compiler eingefügt. Auch wenn die Flexibilität des Programmierers/Modellierers dadurch bewusst eingeschränkt wird, beinhaltet diese Form der Abstraktion keinen wirklichen Informationsverlust. Die Modellierung kontinuierlicher Systeme durch Matlab Simulink-Blockdiagramme etwa ist ein typisches Beispiel für Modellierung, die auf dieser Form der Abstraktion beruht.

Bei der *Abstraktion durch expliziten Informationsverlust* hingegen kann fehlende Information nicht ohne weiteres durch einen Makroexpansionsmechanismus wiederhergestellt werden. Diese Form findet etwa dann statt, wenn Teile der Funktionalität, etwa bestimmte Spezialfälle, im Modell nicht berücksichtigt werden. Das Ignorieren von Zeitverhalten oder anderen QoS-Attributen ist eine andere weit verbreitete Abstraktion dieses Typs. Bei der Erzeugung von Produktionscode oder Testfällen muss die fehlende Information offenbar explizit hinzugefügt werden.

In der Praxis des modellbasierten Testens findet man beide Formen, meist in Kombination. Details (nicht Präzision!) gehen nun überwiegend durch die zweite Form der Abstraktion verloren. Entscheidende Bedeutung kommt deshalb den die Abstraktionsunterschiede überbrückenden Adapterkomponenten zu; Schätzungen gehen von etwa 40% des Gesamtaufwands für deren Implementierung aus [27] (Teile der Adapterkomponenten – Testharnesses – sind allerdings auch bei nicht modellbasierten Testansätzen nötig). Es ist stets zu überprüfen, wie der Aufwand für diese Komponenten im Verhältnis zu den verminderten Aufwendungen bei der Validierung des Modells anstelle des Codes steht.

Zahlreiche proof-of-concept-Studien zum MBT (Annahme 3, s.u.) geben Anlass zu der Vermutung, dass adäquate Modelle stets gefunden werden können. Der zweite Teil der Annahme, der sich auf den verminderten Aufwand einfacherer Artefakte bezieht, erscheint intuitiv erfüllt.

Die mögliche Wiederverwendung (Annahme 5) von Modellen und Testselektionskriterien, insb. Umweltmodellen, basiert auf der folgenden Idee. MBT erhöht den Abstraktionsgrad der Artefakte, mit denen der Tester hantiert: Anstelle von einzelnen Testfällen und Testsuiten wird mit Verhaltensmodellen und Testselektionskriterien operiert. Aus einem Testselektionskriterium werden möglicherweise automatisch ggf. viele Testfälle generiert. Die Idee ist nun, dass Änderungen auf einem höheren Abstraktionsniveau (Modell, Selektionskriterien) leichter durchzuführen und zu validieren sind als auf einem niedrigeren Niveau (Testfälle). Argumentiert wird üblicherweise über die „Lokalität" der Änderung: Eine lokale, leicht durchzuführende Änderung auf Modellebene mit möglicherweise globalen Implikationen ist leichter zu validieren als die entsprechende globale Änderung auf Testsuiteebene. Das wird i.a. von der Art des Validierungsmechanismus abhängen.

Adapterkomponenten, die sowohl den Abgleich der Abstraktionsniveaus vornehmen als auch die Tests durchführen, sind ebenfalls Kandidaten für Wiederverwendung.

Zur Anzahl von Testfällen (Annahme 6) schließlich ist festzuhalten, dass automatische Testfallgeneratoren i.a. natürlich beliebig viele Tests generieren können. Es besteht zunächst aber nicht unbedingt ein Zusammenhang zwischen der Güte einer Testsuite, im Sinn von Anzahl detektierter Fehler, und ihrer Größe. Im Bereich eingebetteter Systeme (etwa Chipkarten, Netzwerkcontroller) kann die Durchführung eines Tests aufgrund von Rüstzeiten etwa 15 Sekunden dauern, was die Maximalanzahl durchzuführender Tests beschränkt. Für Businessinformationssysteme gilt das i.a. natürlich nicht, was die Technologie u.a. für die Erzeugung von Regressionstests z.B. nach Refactorings interessant macht. Durchgeführte Tests müssen außerdem analysiert werden, und wenn aus einer riesigen Testsuite etwa die Hälfte aller Tests auf demselben Fehler basiert, wird die Analyse schnell sehr teuer; es kann dann sinnvoll sein, alle Tests erneut durchzuführen und somit die Anzahl detektierter Differenzen zwischen Soll- und Istverhalten massiv zu reduzieren.

3.3 Diskussion der Annahmen mit publizierter Evidenz

Dass eine explizite Modellierung bei der präzisen Definition von Anforderungen und Systemen (Annahme 1a) hilfreich ist, ist weitgehend unumstritten. Existierende Evidenz wird hier bewusst nicht umfassend zitiert. Im Bereich des modellbasierten Testens sind als Evidenz insbesondere zwei Studien zu nennen [5,24]. Die erste beschreibt verschiedene Stadien des Modells eines Flugsteuerungssystems und zeigt, wie verschiedene QS-Techniken (Reviews, Model Checking, Ableitung von Modelltraces zur manuellen Überprüfung) verschiedene Fehler im Modell aufdecken.

Die zweite zeigt am Beispiel eines Netzwerkcontrollers, wie viele Fehler durch die Modellierung aufgedeckt wurden und insbesondere auch, dass bestimmte Modellierungsfehler erst durch das Ausführen modellbasiert generierter Tests offenbar wurden. In der Praxis der Ausführung modellbasiert erstellter Tests zeigt sich, dass Fehler stets sowohl in der SUT als auch im Modell gefunden werden, was die Notwendigkeit der Redundanz unterstreicht (Abschnitt 2).

Bezüglich der intuitiv einleuchtenden Annahme, dass Modelle neben ihrer Funktion als Spezifikation gewissermaßen als Nebeneffekt auch für die Generierung von Tests (Annahme 1b) verwendet können, sind dem Verfasser keine Dokumentationen bekannt. Ein Grund dafür könnte sein, dass die meisten Studien zum MBT auf Modellen basieren, die für bereits existierende SUTs erstellt wurden.

Die Effektivität des MBT (Annahme 3) in Bezug auf detektierte Fehler, zumeist ohne Berücksichtigung der Kosten und ohne Vergleich mit anderen QS-Techniken, ist eindrücklich dokumentiert. Das liegt daran, dass diverse Forschergruppen zur Bewertung der von ihnen entwickelten Technologie Fallstudien publizieren und in diesen stets Fehler finden konnten [1,3,5,9,10,11,12,15,16,18,20,24,25,17]. Diese Erkenntnisse müssen natürlich mit den Resultaten alternativer QS-Techniken in Bezug gesetzt werden – dass bei sorgfältiger Modellierung eines Systems Probleme gefunden werden, überrascht zunächst nicht (s. den folgenden Abschnitt zur Annahme 4). Dennoch zeigen die Studien klar das Potential des MBT auf.

Die für die industrielle Verbreitung von Techniken des MBT relevanteste Form der Evidenz bezieht sich auf den relativen Aufwand/Nutzen des MBT (Annahme 4): wie aufwendig ist die Erstellung modellbasierter Tests, d.h. von Modell, Selektionskriterien und Adapter, im Vergleich zu herkömmlichen Formen des Tests (zum Vergleich mit anderen QS-Maßnahmen s. andere Arbeiten [13]), und wie unterscheidet sich die „Qualität“ der Tests, z.B. in Bezug auf die Anzahl detektierter Fehler?

Untersuchungen über die Qualität von Tests finden sich in den folgenden Aufsätzen (1,2 berücksichtigen den Nutzen und ignorieren Kosten; 3,4 berücksichtigen auch Kosten).

1. Dalal et al. [11] fassen Erfahrungen aus vier industriellen Projekten zusammen. Ihr Ansatz zum modellbasierten Testen basiert ausschließlich auf Umwelt- bzw. Eingabedatenmodellen (pairwise testing), d.h. es gibt kein automatisiertes Orakel. MBT findet signifikant mehr Fehler als nicht modellbasiertes Testen.
2. Pretschner et al. [24] untersuchen MBT auf der Basis von erweiterten Zustandsmaschinen mit I/O. Die Modellierung findet eine große Anzahl von Fehlern in den vorher vorhandenen Spezifikationen, modellbasierte Tests finden signifikant mehr Fehler als nicht modellbasierte Tests, und automatisch generierte modellbasierte Tests finden nicht mehr Fehler als manuell erstellte modellbasierte Tests.
3. Bernard et al. [3] beschreiben ihren Ansatz mit in B geschriebenen Modellen. Testfälle werden anhand struktureller Kriterien erzeugt. Eine auf Anforderungen (nicht gefundenen Fehlern) basierende Subsumptionsbeziehung zwischen Testfällen wird definiert, und es wird gezeigt, dass automatisch generierte Tests i.a. eine ad-hoc

handerstellte Referenzsuite umfassen. Modellbasiertes Testen ist halb so aufwendig wie herkömmliches Testen.

4. Farchi et al. [15] beschreiben zwei Studien, in denen MBT mit in einer Variante der Murphi-Eingabesprache notierten Modellen Fehler findet, die ohne explizite Modelle nicht gefunden wurden. MBT erfordert einen um 15 Prozentpunkte niedrigeren Aufwand als nicht explizit modellbasiertes Testen.

Schließlich dokumentieren die Arbeiten von Clarke [8] und Blackburn et al. [5] bezüglich sowohl Effektivität als auch Effizienz signifikante Vorteile des MBT gegenüber nicht-modellbasiertem Testen. Da die Diskussion vermutlich aus Vertraulichkeitsgründen stark abstrahiert erfolgt, wird hier auf eine nähere Beschreibung verzichtet.

4 Schlussfolgerungen

Testen ist notwendigerweise modellbasiert [4], ob das Sollverhalten nun implizit gegeben oder in ein explizites Modell codiert ist. Viele bis heute offene Fragen, etwa die nach geeigneten Testselektionskriterien, sind demnach unabhängig davon, ob konventionelles Testen oder die explizit modellbasierte Spielart betrachtet wird. Dieses Papier hat die Erfolgsfaktoren in Form von zumeist impliziten Annahmen und existierende Evidenz benannt, die sich insbesondere auf die kostengünstige und hochqualitative Entwicklung von nicht nur zum Testen verwendeten, hinreichend abstrakten und detaillierten Modellen sowie Testselektionskriterien und Adaptern beziehen.

Die erwarteten Vorteile des MBT lassen sich dann wie folgt zusammenfassen.

1. Die Tatsache, dass das o.g. implizite mentale Modell explizit gemacht wird, erlaubt es, auf strukturierte Art und Weise ausreichend viele und gute Tests bei akzeptablen Kosten zu erzeugen.
2. MBT liefert „bessere“ und „billigere“ Tests als Strategien, die nicht auf expliziten Modellen basieren
3. Der Vorgang der Modellierung an sich sorgt dafür, dass Probleme in den Spezifikationsdokumenten frühzeitig erkannt und behoben werden können. Das MBT erzeugt somit einen zusätzlichen Wert des Modells.

Eingebettete Systeme. Die hier aufgeführten Betrachtungen sind bewusst so allgemein gehalten, dass sie sowohl für den Bereich eingebetteter Systeme als auch für den von Businessinformationssystemen zutreffen. Dennoch gibt es natürlich auch im Bereich des (modellbasierten) Testens spezielle relevante Charakteristika beider Domänen. Erstens ist der Test eingebetteter Systeme insofern erschwert, als der interne Zustand des Systems üblicherweise nicht bekannt ist, was dazu führt, dass spezielle Signalfolgen angelegt werden müssen, um diesen Zustand zu bewerten. Zweitens sind die Qualitätsanforderungen an eingebettete Systeme insofern höher, als Rückrufaktionen von Chipkarten oder automobilen Steuergeräten sehr teuer sein können. In der Konsequenz „darf“ der Testprozess ggf. etwas aufwendiger ausfallen. Drittens ist die Anzahl der Testfälle für eingebettete Systeme bisweilen durch die Hardware eingeschränkt—anstelle von Millionen von Testfällen können ggf. nur Tausende ausgeführt werden. Viertens sind eingebettete Systeme (heute noch) durch einen eher komplexen Kontrollfluss als durch komplexe verwendete Datentypen charakterisiert. Das hat Konsequenzen insofern, als strukturelle Abdeckungskriterien für den ersten Fall besser verstanden zu sein scheinen. Schließlich sind Modelle eingebetteter Systeme und insbesondere Modelle ihrer Umwelt häufig kontinuierlicher oder hybrider Natur, was direkte Konsequenzen für die Technologie der Testfallgenerierung zeitigt.

Vielversprechende Forschungsgebiete. Erste Evidenz für den erfolgreichen und möglicherweise kosteneffektiven Einsatz des MBT ist verfügbar. Die folgenden Fragen harren allerdings noch einer soliden Beantwortung: Wieviele Fehler werden während der Modellierungsphase allein gefunden? Wie effektiv ist die Automatisierung? Wie schneidet MBT im Vergleich mit konkurrierenden Technologien ab? Wie verhalten sich Kosten und Nutzen zueinander, insbesondere unter der Annahme, dass existierende Testmanagementsysteme eingesetzt werden? Welche domänenspezifischen Charakteristika gibt es? Welche organisatorischen und prozessbezogenen Voraussetzungen sind unabdingbar? Die Technologie der Testfallerzeugung (s. die Überblicksarbeiten zu existierenden Werkzeugen [2,27]) ist zumindest für deterministische ereignisdiskrete Systeme ohne harte Echtzeitanforderungen soweit ausgereift, dass empirische Studien zur Beantwortung dieser Fragen bei allen existierenden Schwierigkeiten in Angriff genommen werden können.

Eine weitere relevante offene Frage, die aus dem Feld des MBT herausführt, liegt in der Entscheidung über adäquate Abstraktionsniveaus bei der ingenieurmäßigen Erstellung von Modellen von SUT und Umwelt, unterstützt etwa durch mehr oder weniger domänenspezifische Abstraktionsmuster. Ebenfalls aus dem Gebiet des MBT heraus führen Forschungen, wie Modelle effizient erstellt werden können. Das beinhaltet Versionierungsmechanismen für Modelle, Modell-Debugger, werkzeuggestützte Entwicklungsschritte wir Refactorings und Regressionstests. Die Verknüpfung von Verhaltensmodellen mit Testmiddleware scheint technisch ebenfalls noch nicht befriedigend gelöst. Die Forschung zu Spezifikationssprachen für Selektionskriterien (nicht für die Tests selbst, wie TTCN-3) befindet sich noch in den Anfängen. Schließlich ist bis heute unklar, auf welcher Grundlage Selektionskriterien für konkrete Anwendungen oder Fehlerklassen definiert werden sollen.

Der Verfasser ist überzeugt, dass das MBT eine vielversprechende Technologie darstellt. Die zugrundeliegenden Ideen erscheinen insbesondere dann attraktiv, wenn sie im Zusammenhang übergreifender modellbasierter Entwicklungsprozesse gesehen werden.

Danksagung. B. Köpf, J. Philipps und B. Schätz bin ich für wertvolle Anregungen und Kommentare zu einer frühen Version dieses Aufsatzes dankbar.

5 Literaturverzeichnis

1. A. Belinfante, J. Feenstra, R. d. Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, L. Heerink : *Formal test automation: A simple experiment*. Proc. 12th Intl. workshop on Testing of Communicating Systems, S. 179-196, 1999
2. A. Belinfante, L. Frantzen, C. Schallhart: *Tools for Test Case Generation*. [6], S. 391-438

3. E. Bernard, B. Legeard, X. Luck, F. Peureux: *Generation of test sequences from formal specifications: GSM 11.11 standard case-study*, SW Practice&Experience 34 (10):915-948, 2004
4. R. Binder: *Testing Object-Oriented Systems: Models, Patterns&Tools*. Addison Wesley, 1999
5. M. Blackburn, R. Busser, A. Nauman: *Why model-based test automation is different and what you should know to get started*. Proc. Intl. Conf. on Practical SW Quality and Testing, 2004
6. M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, A. Pretschner (Hrsg.): *Model-Based Testing—a tutorial volume*. Springer LNCS 3472, 2005
7. O. Buehler, J. Wegener: *Automatic testing of an autonomous parking system using evolutionary computation*. SAE world congress, 2004.
8. J. Clarke: *Automated Test Gen. from Behavioral Models*. Proc.11th SW Quality Week, 1998
9. D. Clarke, T. Jéron, V. Rusu, E. Zinovieva: *Automated Test and Oracle Generation for Smart-Card Applications*. Proc. E-smart, S. 58-70, 2001
10. I. Craggs, M. Sardis, T. Heuillard: *AGEDIS Case Studies: Model-Based Testing in Industry*. Proc. 1st Eur. Conference on Model Driven Software Engineering,S. 129-132, 2003
11. S. Dalal, A. Jain, N. Karunanithi, J. Leaton, C. Lott, G. Patton, B. Horowitz: *Model-based testing in practice*. Proc. ICSE'99, S. 285-294, 1999
12. J. Dushina, M. Benjamin, D. Geist: *Semi-formal test generation with Genevieve*. Proc. 38th conf. on Design automation, S. 617-622, 2001
13. A. Endres, D. Rombach: *A Handbook of Software and Systems Engineering*. Add Wes, 2003
14. M. Fagan: *Reviews and Inspections*. In Software Pioneers—Contributions to Software Engineering, Springer, S. 562-273, 2002
15. E. Farchi, A. Hartman, S. Pinter: *Using a model-based test generator to test for standard conformance*. IBM Systems Journal 41 (1):89-110, 2002
16. L. Fournier, A. Koyfman, M. Levinger: *Developing an Architecture Validation Suite: Application to the PowerPC Architecture*. Proc. 36th ACM Design Automation Conference, S. 189-194, 1999
17. M. Horstmann, W. Prenninger, M. El-Ramly: *Case Studies*. [6], S.439-461
18. H. Kahlouche, C. Viho, M. Zendri: *An industrial experiment in automatic generation of executable test suites for a cache coherency protocol*. Proc. IFIP TC6 11th Intl. Workshop on Testing Communication Systems, S. 211-226, 1998
19. J. Musa: *Software Reliability Engineering*. Author House, 2. Auflage, 2004
20. J. Philipps, A. Pretschner, O. Slotosch, E. Aiglstorfer, S. Kriebel, K. Scholl: *Model-based test case generation for smart cards.* Proc. 8th Intl. Workshop on Formal Methods for Industrial Critical Systems, S. 168-192, 2003
21. W. Prenninger, A. Pretschner: *Abstractions in Model-Based Testing*. ENTCS 116:59-71, 2005
22. A. Pretschner: *Model-Based Testing in Practice*. Proc. FM'2005, S. 537-541, 2005
23. A. Pretschner, J. Philipps: *Methodological Issues in Model-Based Testing*. [6], S. 281-291
24. A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, T. Stauner: *One Evaluation of MBT and its Automation*. Proc. ICSE 2005, S.392-401
25. J. Shen, J. Abraham: *An RTL Abstraction Technique for Processor Micorarchitecture Validation and Test Generation*. J. Electronic Testing: Theory&Application 16 (1-2):67-81, 1999
26. H. Stachowiak: *Allgemeine Modelltheorie*. Springer, 1973
27. M. Utting, A. Pretschner, B. Legeard: *A Taxonomy of Model-Based Testing*. Eingereicht bei J. Information and Software Technology, 2005
28. G. Walton, J. Poore: *Generating transition probabilities to support model-based software testing*. Software: Practice and Experience 30(10):1095-1106, 2000
29. H. Zhu, P. Hall, J. May: *Software Unit Test Coverage and Adequacy*. ACM Computing Surveys 29:366-427, 1997

Real-time Operating Systems for Self-coordinating Embedded Systems

Franz J. Rammig, Marcelo Götz, Tales Heimfarth, Peter Janacik, Simon Oberthür
Heinz Nixdorf Institut
Universität Paderborn
Fürstenallee 11
D-33102 Paderborn
{franz, mgoetz, tales, pjanacik, oberthuer}@uni-paderborn.de

Abstract: It can be observed that most technological artefacts are becoming intelligent "things that think" and most of these intelligent objects will be linked together to an "Internet of things". To master this omnipresent virtual "organism", completely new design and operation paradigms have to evolve. In this paper we discuss how research of our group at the University of Paderborn is providing fundamental principles, methods, and tools to design real-time operating systems for this virtual "organism" of the future. Based on our fine-granular library for the construction of reflexive RTOS, the necessary configuration tool and its on-line version are discussed. Next step towards self-coordination is a profile management system to support self-optimization of the RTOS. The included flexible resource manager allows migration of RTOS services dynamically between programmable processors and re-configurable HW. In a final step the RTOS itself can be distributed. Its services are provided by a cluster of instances instead of a single one. This makes a sophisticated dynamically self-optimizing communication system necessary.

1 Introduction

In the world of information technology it is no longer the computer in the classical sense where the majority of IT applications are executed; computing is everywhere. More than 20 billion processors have already been fabricated and the majority of them can be assumed to still be operational. At the same time virtually every PC worldwide is connected via the Internet. This combination of traditional and embedded computing creates an artefact of a complexity, heterogeneity, and volatility unmanageable by classical means. Metaphorically speaking, it may be treated as an organism made up of computers, networks, system software, applications, and, most importantly, human beings. This virtual organism as it exists today is still in a state of adolescence. Each of our technical artefacts with a built-in processor can be seen as a "Thing that Thinks", a term introduced by MIT's Thinglab. It can be expected that in the near future these billions of Things that Think will become an "Internet of Things", a term originating from ETH Zurich. This means that (using the above metaphor) we will be constantly surrounded by a virtual "organism" of Things that Think. This organism cannot be managed by any traditional means. As a consequence, completely new RTOS approaches are needed. The virtual organism is volatile by nature. Its constituents are mobile and adapt to changing environmental contexts in a self-organizing manner. This means that the RTOS itself has to be self-adapting as well, which includes that it has to be reflexive. Techniques are needed to allow such an RTOS to sense its environment, reflect this information at its present state, decide about self-modifications and execute this self-modification. Some approaches developed in our group to deal with these challenges will be sketched out in this contribution. These approaches are based on the assumption that system services are provided by means of alternative profiles. An intelligent profile manager selects the presently best suited collection of services and their respective profiles. Re-configurable hardware is commercially available and introduces an additional degree of potentials and at the same time of volatility. Systems consisting of programmable microcontrollers and re-configurable HW can be tailored dynamically towards actual demands by application software. It depends on the actual characteristics of application load whether processor capacity or FPGA area are the more valuable good. This means that RTOS services should migrate dynamically between SW and HW implementation. An additional challenge arises if the global interaction of all these objects is envisioned. Together with potential mobility of the devices, a degree of volatility is introduced that is unknown in traditional systems. Centralized approaches seem to be completely inadequate to cope with this challenge. We therefore developed a cross-layer communication system that uses decentralized techniques to provide a communication backbone in highly volatile environments. These techniques are inspired by the behaviour of ant colonies. If such a self-adapting and self-optimizing communication backbone has been established, a new perspective on the RTOS becomes possible. There may be no longer a need to provide all potentially requested services by each instance of the RTOS on the various nodes. In principle it would be sufficient to provide the services by a complete cluster of instances as a whole. However careful tailoring of service distribution is necessary as any remote service provisioning is very costly due to the extremely high power consumption of communication.

2 Related Work

Some work on customization and configuration as approach for adapting a system to the requirements of an application has been reported in literature. In the field of (real-time) operating systems (OS) various approaches for customization have been proposed. SYNTHESIS [Mas92] adapts its code by partially evaluating and recompiling condition statements depending on available input data at run time. This can result in changing compare instructions and conditional jumps by unconditional jumps and vice versa. Likewise, the OS K42 [SAH+03] includes system support for online reconfiguration by a hot-swap mechanism using C++ virtual function tables.

Reconfigurable hardware/software based (hybrid) architectures are being very attractive for implementation of run-time reconfigurable embedded systems. Moreover, the hardware-software allocation of application tasks to dynamically reconfigurable embedded systems (by means of task migration) allows for customization of their resources during run-time to meet the demands of executing applications, as can be seen in [HMM04] and [MNC+03].

Nevertheless, the current research efforts spent in this field are just focused on application level, leaving to the RTOS the responsibility to provide the necessary mechanisms and run-time support (see [BJS04], [WK01] and [NCV$^+$03]). In [BJS04], e.g., the resources are dynamically arranged to support dynamic application tasks arriving into the system. However, the RTOS itself is still static.

In our novel proposal, we expand and adapt those concepts to the RTOS level. Thus, not just the application but also the RTOS itself may also be reconfigured over a hybrid architecture.

Current OS are not able to work on top of ad hoc networks in a distributed manner (e. g. *VxWorks* [Sys99], *Apertos* [Yok92]). OS for sensor networks like *TinyOS* [ea00] do not support complex, distributed applications using elaborate OS services like our approach.

Several of our approaches are based on the *division of labour in ants* [BDT99]. Conducted experiments suggest that resilience of a colony as a whole is determined by the *elasticity* observed at the individual level, which is modelled as follows: *Intensity of stimulus s* associated with a particular task, determined by quantitative cues sensed; *response threshold θ*, determining the tendency of an individual to respond to a stimulus. Finally, n describing the *steepness* of the threshold. They are combined in the *response function*: $T_\theta(s) = \frac{s^n}{s^n+\theta^n}$.

3 Basic Principles of the DREAMs Library and the TereCs Configuration System

OS and run-time platforms even for heterogeneous processor architectures can be constructed from customizable components (*skeletons*) out of the DREAMS's (**D**istributed **R**eal-time **E**xtensible **A**pplication **M**anagement **S**ystem) library [Dit99]. By creating a configuration description all desired objects of the system have to be interconnected and afterwards fine-grained customized. The goal of that process is to add only those components and properties that are required by the application. The creation of a final configuration description for DREAMS was automated during the project TEReCS (**T**ools for **E**mbedded **Re**al-Time **C**ommunication **S**ystems) [Bök03]. During that project a methodology was developed in order to synthesize and configure the OS for distributed embedded applications.

TEReCS distinguishes between knowledge about the application and expert knowledge about the customizable OS. Knowledge about the application is considered as a requirement specification. This specification is input to the configurator and abstractly describes the behavior of the application and some constraints (deadlines), which have to be assured. The behavior of the application is defined by the OS calls it requests.

The complete and valid design space of the customizable OS is modelled by an AND/OR service dependency graph in a knowledge base. This domain knowledge contains options, costs, and constraints and defines an over-specification by containing alternative options. The configuration process removes some domain specific knowledge by exploiting knowledge about the application. Thereby, a configuration for the run-time platform will be generated. In the AND/OR graph nodes represent *services* of the OS and are the smallest atomic items, which are subject to the configuration. Mandatory dependencies between services are specified by AND edges, optional or alternative are specified by OR edges. Services and their dependencies have costs and can be prioritized. *Constraints* for the alternatives can be specified. Root nodes of the graph are interpreted as *system primitives/system calls* of the OS. Each of these primitives points to one concrete service. The service dependencies span a complete graph. The leaf nodes can refer to hardware devices. These devices are communication devices, which again refer to communication media.

4 Self-Optimization

For building an RTOS system, which optimizes itself and provides the released resources optimally to the applications, we have to answer the three following questions: First, how can we get information about the dynamic resource requirements during run-time of an application? A suitable interface between the application and the OS is required. Second, how can we describe the design space for reconfiguration of the RTOS system? A model is required that describes the dependencies between the system services. Third, based on this information: Which is an adequate system configuration? A resource management system is required to activate/deactivate the system services and to release resources for the applications.

Our self-optimizing RTOS consists of three components Profile Framework, Online TEReCS and Flexible Resource Management (FRM), which are briefly described in this section. A concrete example and a prototype are provided in [OBG05].

Profile Management

The Profile Framework, described in detail in [OB04], has two main purposes: The first purpose is to model information about the dynamic resource requirements of the application and of the system services. The knowledge about the resource requirements of the applications is necessary to optimize the system by deactivating services that are currently not required. The resource requirements of the system services are necessary to determine the amount of resources, which can be released and to guarantee that applications and system services get their required resources under hard real-time conditions. The second purpose is to provide released resources from deactivated system services to the applications. By means of this framework the developer can define a set of profiles per application. Profiles describe different *service levels* of the application, including different quality and different resource requirements. Additionally each profile holds information about the time to activate and deactivate the profile. All resource allocations of an application require an announcement to the FRM. The *maximal assignment delay*, specified in each profile per resource, is the worst case time the assignment of the resource can be delayed from the announcement until its allocation.

Online Configuration System: Online TEReCS

The Online TEReCS component has knowledge of the design space of the customizable components, models system services as profiles, and is responsible for the low-level reconfiguration of the system services. For the online case of the configuration a coarse grained hierarchy of this graph is used on the system service level – one service can be composed of many components. Thus, the overall decision space is more restricted and the number of generated profiles is easy to manage.

For a system service, which can be in a deactivated or in an activated state, two profiles are created. One profile for the activated and one profile for the deactivated state. In the profile for the activated state the service specifies the required resources to fulfill its task. In the profile for the deactivated state the service occupies the resources it provides in the activated state. Hereby, it is guaranteed that the services can only be deactivated if the provided services are not required.

Flexible Resource Manager

The major goal of the FRM [OB04] is to optimize the resource utilization and the over-all system quality by selecting profiles for activation under the actual conditions. To maximize the utilization the FRM puts the resources, which are held back for worst case resource requirements of an application, at other application's disposal. Normally, an application acquires as many resources as it requires for worst-case scenarios. Thus, the application has always enough resources for its tasks and can fulfill its service at any time. These resources are only required when the worst-case scenario occurs. In our approach the FRM tries to minimize this internal waste of resources, by making these resources temporarily available to other applications under hard real-time conditions. The FRM is responsible for switching between the profiles of the applications and system services under the defined switching conditions. The decision is made based on a quality function, which considers the actual resource requirements, the importance of an application, and the quality of the profiles. To guarantee hard real-time conditions, an acceptance test for profile activation is integrated in the optimization process.

5 Reconfigurable Hardware/Software-based RTOS

Differently from the normal approach where the design of such RTOS is done offline, the proposed approach suggests the use of new reconfigurable architectures (e.g. Virtex-II Pro$^{\mathrm{TM}}$FPGA) in order to support the development of a hardware/software reconfigurable OS [Göt04]. In this proposed architecture, the Real-Time OS (RTOS) is capable to adapt itself to current application requirements, tailoring the RTOS components for this purpose. Therefore, the system continuously analyzes the requirements and reconfigures the RTOS components at the hybrid architecture optimizing the use of system resources. Hence, the system is capable to decide on-the-fly which RTOS components are needed and also to which execution environment (CPU or FPGA) they should be assigned. Our system concept is based on a microkernel approach, where the RTOS services are provided as a set of hardware and software components. These components can, during run-time, be reallocated (reconfigured) over the hybrid architecture. The usage of microkernel also incorporates the nature advantage of flexibility and extensibility (among others) of it, which is very desired in our case in order to better perform the reconfigurability aspects.

RTOS Service Assignment

The decision of which RTOS service is to be placed in hardware or software, is based on the amount of resources and the communication costs needed by each service. If an RTOS component is allocated at the software environment, it will usually increase the processor load. Otherwise, at the hardware environment it will require some amount of circuit area available. Similarly, the communication requirements of each service will infer in different communication costs depending on its placement. To solve this assignment problem, a cost function has been proposed [GRP05], which tries to minimize the total amount of resource used by the RTOS component set currently needed by the application. Besides the limited amount of resources available for the RTOS, a weighted load balancing of the resources used is specified as a system constraint. Thus, it is possible to influence the placement decision based on the current valuable goodness of the resources required, since the resources are shared between RTOS and application software. Moreover, through a correct load balancing, it can be avoided that one execution environment gets near to its full utilization and so, making the system capable to absorb variations of the application demands.

System Reconfiguration

As the application requirements are considered to be dynamic, the component costs are not static and depend on the current application demands. This leads to the fact that a certain system configuration (component allocation) may no longer be valid after application changes. Therefore, a continuous evaluation of the components partitioning is necessary. Whenever the system reaches an unbalanced situation, the RTOS components should be reallocated in order to bring the system again into the desired configuration. In this situation, not just the new assignment problem needs to be solved again, but also the costs (time) necessary to reconfigure the system from the current state to new one need to be evaluated. This evaluation is necessary since we are dealing with real-time systems and so, it has a limited time available for reconfiguration activities (which depends on the current application time constraints). For this case, our approach consists on scheduling the RTOS components reconfiguration together with the application tasks. Therefore, techniques based on RTOS scheduling theory, like fixed/dynamic priority servers, are used to provide a safely time for reconfiguration activities and still guarantee the correct application tasks timeliness.

6 Self-optimizing Communication System

In the introduced virtual “organism” made up of mobile constituents, wireless communication plays an important role. It is the logical next step after having considered wired communication in our previous work, which is only suited for static networks. Given its relatively high cost, the means of wireless communication has to be utilized economically.

Employed within this context, self-organization promises lower complexity and communication overhead, as well as, increased robustness and good scalability properties. We aim to use this powerful paradigm at all levels of the network stack. Our efforts are predominantly directed towards the following aspects: topology control, combined link metrics, as well as, adaptive ant colony-based multipath routing.

Network regions with higher node density tend to lead to a greater amount of energy waste through overhearing, collisions, and retransmissions after frame losses, while not increasing the network's capacity. Therefore, we develop an emergent *topology control* reducing the number of active nodes in such areas. It divides nodes in *transporters*, actively participating in network operation (e. g. part of a routing backbone), and *workers*, not involved in these activities. The underlying mechanism for state transitions is motivated by the *division of labour in ants*. State changes are realised using a response function, as reviewed in Section 2. The underlying idea behind the response function for the *worker-to-transporter transition* is to use a threshold function that decreases the tendency for a worker to become a transporter with more energy consumed and a lower density of neighboring worker nodes. The stimulus rises with a longer disconnection time from the backbone. On the other hand, the underlying idea of the response function for the *transporter-to-worker transition* is to use a threshold that decreases, when the backbone around a transporter becomes denser. The stimulus rises with the amount of energy consumed since becoming transporter.

Efficient utilisation of the wireless medium can only be realised given a good estimate of the "real" quality (in terms of the expected reception success) of the underlying links. Based thereon, protocols and applications at upper layers are given the means to adjust their optimization strategies. None of the basic, conventional metrics provides fully satisfying results: Received signal strength measurements are rough, but quick estimates. The observation of transmission successes yields precise results, but only after enough data is provided. Neither of these metrics recognises volatile links, so that upper layers could avoid them. Hence, we use a *combined link metric*, the so called *virtual distance*, incorporating received signal strength, reception success rate, and a history component.

At the network layer, we extend state-of-the-art *ant colony-based multipath routing* by several elements. Our choice for ant colony-based routing was motivated by its promising characteristics: utilisation of multiple paths, probabilistic decisions, consideration of lower-layer feedback, and local work, helping to achieve more resilience, a higher degree of load balancing, and a lower communication overhead. To accelerate the transitory phase in ant-colony routing, we propose a novel *waycost function* enabling pheromone adjustments that more accurately reflect underlying properties. To enable this, packets carry a field containing the combined link metric-based, accumulated waycost taking into account all visited (i.e. utilized) links. The amount of pheromone deposited on a packet's way then decreases exponentially with increasing waycost. In order to reduce the frequency of expensive route failures and reconstructions due to changes incurred by topology control, we further propose a fuzzy distinction between transporters and workers: during a next-hop decision, the probability for choosing a transporter is increased.

In ns-2 [The] simulations, we demonstrated the scalability in terms of node number and density, as well as, the stability of the results of topology control. Further, the results showed a significant improvement of the quality of utilised links using the combined link metric.

7 NanoOS – Service Provisioning by Clusters of OS Instances

In an “Internet of things” scenario, heterogeneous sorts of devices are connected through different network infrastructures. Vantages can be achieved with this scenario with the introduction of distributed computing instead of bare centralized one. We can generalize the idea of self-optimization from one node to a network of connected devices. In our approach, we are considering small devices connected through an ad hoc network. In order to support complex functionality in constrained nodes, the OS uses a novel approach: it distributes the services among the nodes. This means that each node of the system has just a small part of the kernel of the complete OS; a group of nodes together form an instance of the OS. Thereby the resource requirements in each node are much less than on a node with all services locally instanced. The prototype of an OS, built over the self-optimizing communication system presented in the previous section, enables this vision. It is called NanoOS [HR04].

In the NanoOS, each application has one goal and is composed by a set of tasks; tasks are the atomic unit of the distributed application. The developer is responsible for the division of the application in several cooperative tasks. For this the OS supports hierarchic grouping of tasks. The services and tasks can migrate in order to optimize the communication. Each service is autonomous, it is an agent that tries to find a good position in the network, a node with enough resources and where the communication with the clients (application tasks) is minimized.

Resource-aware Clustering

The distribution of the services brings several problems. The consistency of the kernel in each node should assure also a global consistency of the entire “distributed” kernel (global consistency). The connection quality plays an important role in this case. In order to restrict the traffic between nodes with good connection quality and also to reduce significantly the organization overhead of the OS (small routing table, service discovery scope and consistency control), the entire system is divided in clusters. Each cluster should contain the necessary resource for all services running inside the cluster. This brings the concept of a complete OS instance per cluster. ***Cluster***: set of nodes where all the services required by the tasks in this set are available in nodes inside the set.

We developed a heuristic that uses just a small amount of local communication to find the clusters that contain a minimum amount of resources. As result of our clustering algorithm, each node will be assigned to a cluster $\phi \in 0, ..., n-1$ where n is the number of clusters. During the clustering, a node can be of type *clusterhead* (task that represent a cluster), *member* (ordinal member of a cluster) or *not clustered* (task that does not belong to any cluster). The main idea of the algorithm is to use the principle of the stimulus/threshold found in the division of labor of some social insects: members of a nest with different morphology have different threshold to do a determined task. In our case, nodes with good communication and plenty neighborhood have a smaller threshold to become *clusterhead* than nodes with few loosely connected neighbors. The stimulus to become clusterhead increases with the time that a node is of *not clustered* type.

Service Distribution

The distribution algorithm is responsible for deciding where each service of the OS should be placed inside a cluster. The algorithm is executed in each service that is independent to decide the destination node. The main goal of the distribution algorithm is to reduce the global communication overhead (in a continuous self-optimization process), therefore the state of the wireless links is an important parameter used in the optimization process. For this reason, we used the presented link metric that rates the connection quality of the ad-hoc links (*virtual distance*). The positioning of the services is done based on this virtual distance graph and the resource availability in each node. It tries to minimize the sum of all virtual distances of all remote system calls. This is an *NP-Complete* combinatorial problem with similarities with the QAP (*Quadratic Assignment Problem*). To solve it distributively, we developed an heuristic based on the principles of the *swarm intelligence*, more specifically, *Ant Colony Optimization* [BDT99]. Each service of the OS is modeled as a food source and the calls coming from different nodes are the ants looking for the service. Each node has a pheromone table. In each node where an ant passes when going in the direction of the food, it increases the pheromone of correspondent service in the same proportion to the communication cost of the request. The optimization is done when the service (food source) **moves** in the direction of the higher pheromone; that means the service migrates in the direction from where more calls are coming.

8 Conclusion

With the library for real-time operating services called *DREAMS* we are providing a sound basis for the construction of tomorrow's real-time operating systems. We extended this fundamental component-based library by means of a configuration system that can be applied even in on-line mode. Strategic decisions are made by a profile management system, supported by a flexible resource manager. Our reflexive RTOS is able to migrate services dynamically between SW and HW implementations to make sure that always the actually most valuable resources are available for application processes. A fully decentralized, self-organizing communication system for ad-hoc networks, based on ant colony inspired techniques opens the path to highly distributed, highly dynamic systems. This communications system allows even to distribute RTOS services over a cluster of instances of the RTOS in such a way that the total functionality of the RTOS is no longer provided by each individual instance but by the cluster in total.

In the future this innovative approach will be extended further. More sophisticated learning algorithms and even more biologically inspired techniques will be included. The solution will be applied in a variety of applications ranging from sensor networks via cooperative mini-robots up to swarm-based space missions.

References

[BDT99] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems.* Oxford University Press, New York, NY, 1999.

[BJS04] Krishnamoorthy Baskaran, Wu Jigang, and Thamipillai Srikanthan. Hardware Partitioning Algorithm for Reconfigurable Operating System in Embedded Systems. In *Sixth Real-Time Linux Workshop*, 2004.

[Bök03] C. Böke. *Automatic Configuration of Real-Time Operating Systems and Real-Time Communication Systems for Distributed Embedded Applications*. Phd thesis, Paderborn University, Paderborn, Germany, 2003.

[Dit99] C. Ditze. *Towards Operating System Synthesis*. Phd thesis, Paderborn University, Paderborn, Germany, 1999.

[ea00] J. Hill et al. System architecture directions for networked sensors. In *Proc. of the 9th int. conf. on architectural support for programming languages and OS*, 2000.

[Göt04] Marcelo Götz. Dynamic Hardware-Software Codesign of a Reconfigurable Real-Time Operating System. In *ReConFig*. Mexican Society of Computer Science, 2004.

[GRP05] Marcelo Götz, Achim Rettberg, and Carlos E. Pereira. Towards Run-time Partitioning of a Real Time Operating System for Reconfigurable Systems on Chip. In *Proc. of IESS*, Manaus, Brazil, 2005.

[HMM04] J. Harkin, T. M. McGinnity, and L. P. Maguire. Modeling and optimizing run-time reconfiguration using evolutionary computation. *Trans. on Embedded Computing Sys.*, 2004.

[HR04] T. Heimfarth and A. Rettberg. NanoOS - Reconfigurable OS for Embedded Mobile Devices. In *In Proc. of the International Workshop on Dependable Embedded Systems*, Florianopolis, Brazil, 2004.

[Mas92] H. Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. Phd thesis, Columbia University, 1992.

[MNC$^+$03] Jean-Yves Mignolet, Vincent Nollet, Paul Coene, Diederik Verkest, Serge Vernalde, and Rudy Lauwereins. Infrastructure for Design and Management of Relocatable Tasks in a Heterogeneous Reconfigurable System-on-Chip. In *DATE*, 2003.

[NCV$^+$03] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Designing an Operating System for a Heterogeneous Reconfigurable SoC. In *Proc. of IPDPS*, 2003.

[OB04] S. Oberthür and C. Böke. Flexible Resource Management - A framework for self-optimizing real-time systems. In B. Kleinjohann, Guang R. Gao, H. Kopetz, L. Kleinjohann, and A. Rettberg, editors, *Proc. of IFIP Working Conference on Distributed and Parallel Embedded Systems (DIPES'04)*. Kluwer Academic Publishers, 23 - 26 August 2004.

[OBG05] S. Oberthür, C. Böke, and B. Griese. Dynamic Online Reconfiguration for Customizable and Self-Optimizing Operating Systems. In *Proc. of the 5th ACM international conference on Embedded software (EMSOFT'2005)*, pages 335–338, 18 - 22 September 2005. Jersey City, New Jersey.

[SAH$^+$03] C. Soules, J. Appavoo, K. Hui, D. Silva, G. Ganger, O. Krieger, M. Stumm, R. Wisniewski, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis. System Support for Online Reconfiguration, 2003.

[Sys99] Wind River Systems. VxWorks 5.4 - Product Overview, June 1999.

[The] The network simulator. Online. http://www.isi.edu/nsnam/ns/, accessed April 8, 2005.

[WK01] Grant Wigley and David Kearney. The Development of an Operating System for Reconfigurable Computing. In *FCCM*, 2001.

[Yok92] Y. Yokote. The Apertos Reflextive Operating System: The Concept and Its Implementation. In *OOPSLA Proceedings*, 1992.

Some motivation and current results for synchronous modeling and implementation

Jan Romberg

Abstract: Synchronous modeling and programming has been around for roughly two decades now, and has traditionally been associated with design of hardware, abstract control algorithms, and non-distributed software. In this paper, we argue that synchronous modeling is a practically viable option for model-based design and implementation: we'll list some evidence that, for the embedded sector, synchronous techniques have some superior characteristics over, for instance, asynchronous UML-based modeling techniques. We also argue that the true potential of synchronous techniques for construction of larger, concurrent systems has not been fully realized to date: to that end, some current advancements for implementing synchronous programs in distributed settings are described.

1 Motivation: Why synchronous?

In embedded systems design, there is obviously a need for varying modeling paradigms and degrees of precision at different development phases. The spectrum ranges from informal, weakly structured, largely textual requirements models to detailed behavioral models of a system. In this paper, we shall restrict ourselves to the "lower" end of this spectrum: detailed, behaviorally defined models of embedded control systems, which may serve as a basis for model-based verification and validation. The first question we shall examine is: *What is an adequate way of modeling, i.e. expressing, a design for an embedded system at this level?* As a part of the answer, we will identify some weak spots in current asynchronous UML-based methods, which are favorably solved by synchronous techniques. Our treatment will by no means cover all relevant aspects, and will be heavily biased towards concurrency and timing issues. The second question considered is: *Are synchronous models implementable in realistic settings?* We'll present some current results that allow some optimism in this respect, and identify some open issues for research.

1.1 Comparison: The contestants

Synchronous modeling in a nutshell. Synchronous programming and modeling techniques have been popularized by languages such as STATEMATE, ES-

TEREL, or the discrete-time subset of SIMULINK. Our own work includes contributions to the AutoFOCUS tool, which also uses a synchronous model of execution. The characterizing features of synchronous models relevant to this discussion is the *existence of a global, discrete time*, *statically boundable resource consumption*, and the *determinism of computation and communication* with respect to the timebase. The first feature is quite intuitive and probably needs no further explanation. In a common time grid, both aperiodic and periodic computations of varying frequency can be represented by using special symbols to indicate absence of messages. Advanced compiler features, such as the clock calculi of LUSTRE, SIGNAL [BCE$^+$03], and AutoFOCUS [BBR$^+$05], may be employed for both comfortable and consistent modeling of multiclock designs. The second feature, statically boundable time and memory consumption, follows from (1) the (frequent) restriction of individual processes to the finite-state, finite-iteration case, and (2) the restriction to length-preserving processes on the semantics level, so composition preserves finiteness. The third feature, determinism of computation and communication, implies among other things that the timing of message emission and consumption is fixed with respect to the logical timebase.

A sparring partner: UML-RT As the other subject of our comparison, we use the notation formerly known as "Real-Time UML", now subsumed in the UML 2.0 standard [OMG05]. For the sake of argumentation, we use the semantics of state machines and communication realized in IBM's Rational RealTime and Telelogic Tau tools as a reference implementation of the UML standard. These tools are both popular, and share many characteristics with other real-time UML tools evaluated in a case study [SRS$^+$03]. The current UML standard itself [OMG05] is too vague about semantic issues for an adequate assessment. Because there is no "official" published semantics for the above tools either, a rough characterization of the semantics implemented by the respective simulators is considered here, which we'll call *singlequeue UML-RT*.

Singlequeue UML-RT uses no predefined time grid for message emission and consumption, giving it an "asynchronous" or "untimed" flavor. Messages emitted from individual objects, or capsules, are stored in a single[1] global queue which is common to all capsules of the design. The FIFO property of the global queue effectively ensures the absence of race conditions on the simulation level: for any pair of messages, the mutual order of consumption is equal to the order of emission.

1.2 Three criteria for "good" models

According to [Sel03], the success of model-based engineering is bound to criteria such as *abstraction*, *understandability*, and *predictiveness*, among others. The author of [Sel03] claims that the popular UML 2.0 standard already provides adequate

[1] There are UML-RT variants which use different priority levels for messages, resulting effectively in several global queues. For simplicity, we shall not consider this case here.

mechanisms with respect to these three criteria. Upon close inspection, however, we shall argue that at least the singlequeue approach for UML-RT has weaknesses with respect to abstraction, understandability, and predictiveness. This is especially true when concurrency and timing issues are considered.

Abstraction and predictiveness Abstraction is useful if details not relevant for a certain purpose (e.g. making an implementation-independent design) can be hidden from an abstract representation. For fidelity of this representation, it is essential that the abstract model is also *predictive*, i.e. results or properties obtained on an abstract level will be preserved all the way to an implementation in an understandable way. In our opinion, this should be especially true for the *behavior* and *timing* of the model.

The simulation semantics of singlequeue UML-RT models, however, is not guaranteed to correspond closely with the actual semantics of the implementation. Some examples:

Races and unpredicted behavior. The simulation semantics of UML-RT is very costly to implement in distributed settings: it necessitates a distributed agreement on the global message order. Consequently, tools such as Rose RealTime or Telelogic Tau use a cheaper local-queue strategy for distributed implementation. The corresponding (non-deterministic) semantics on the implementation level introduces unpredicted behaviors to the implementation. For instance, in multithreaded implementations of UML-RT models, the relative order of subsequent messages, as seen by a receiver in the simulation, is not guaranteed to be preserved. As a consequence of such race conditions, the resulting behavior may be qualitatively different. For instance, the implemenation may reach a state that is not reachable in the "abstract" model, and vice versa.

Unknown timing and memory consumption. The overall time and memory consumption of a model is undecidable in UML-RT, and may overrun any bound for some models. Tools and simulators usually make no attempt at predicting resource consumption, and a particular timing or scheduling order observed in the simulation may or may not correspond to the implementation. As a consequence, Dohmen and Somers [DS02] conclude that "UML-RT does not support the [...] implementation and verification of hard realtime constraints." [2]

In synchronous modeling, on the other hand, the timing of computations and message transit is clear from the semantics and an appropriate refinement mapping, and relative timing of message arrival is unambigously defined in the model. The

[2] There are a number of academic contributions for real-time analysis of restricted UML models, such as [SPFR98], which typically rely on a fixed mapping of capsules/objects to tasks, and a fixed assignment of task priorities. It is questionable, however, whether this additional mapping information should be considered part of the "model", as it would severely limit implementation-independence and portability of the design, especially in distributed settings.

model semantics yields obligations for the implementation, e.g. allowable execution times and message delays, that an implementor has to prove. Granted, the timing and concurrency assumptions inherent in synchronous models are rather strong, and may be in conflict with some platform-related considerations in cost-sensitive domains such as automotive. For instance, worst-case execution time estimation for modern processors is hard, and if estimates are made too conservative in order to safely implementing the synchronous semantics, the goal of having high processor utilization (and building a cost-effective system) is compromised.

Understandability In order to be effective and understandable, modeling techniques should be as close to the domain as possible, while offering a clean and concise set of modeling concepts. In this context, we will take a quick look at two example issues: *(discretized) continuous controller design*, and *communication/concurrency primitives*.

Continuous controller design. UML-RT is generally not considered strong in the design of continuous controllers, as the notation was originally conceived with very different applications in mind.

Consequently, for the frequent case of continuous and discrete/continuous controller design, UML-RT is not very popular as a modeling notation. For the automotive sector, this fact is documented by the high market share of data-flow tools such as ASCET or SIMULINK (roughly 75%) vs. other tools reported in a recent market study [Han03]. Technically, the integration of block diagrams and data-flow algorithms for control laws with asynchronous UML-RT poses some issues, as control laws are inherently programmed with respect to a global, synchronous timebase.

In comparison, synchronous programming is a suitable paradigm both for discrete controller modeling (e.g. STATEMATE, ESTEREL) and continuous controller modeling (e.g. SIMULINK).

Communication/concurrency primitives. Being based on sequential programming languages, UML-RT offers several different constructs for concurrency, control flow, and communication. For instance, in UML-RT, two objects may call each other through synchronous method calls. Special kinds of objects (called "capsules" in UML-RT) may also call each other through asynchronous messages. These different options for communication and composition can be problematic: they typically encourage developers to encode implementation-specific considerations in the model. For instance, a developer may choose to model an client/server-style interaction between two objects as a synchronous method call. This seemingly minor decision ties the two objects together for the implementententation; once the decision has been made, replacing the synchronous call by a distributable transaction is usually very hard and time-consuming.

Synchronous modeling languages, on the other hand, can usually restrict themselves to a very limited set of constructs for communication and composition. These constructs, such as messages, are typically implementation-independent. For instance, for communication, discrete-time SIMULINK only distinguishes between "signals" for data flows, and "triggers" for events. AutoFOCUS, and other synchronous languages, do not even distinguish event and data traffic.

2 Implementations of synchronous models

Because synthesis of implementations from behavioral models is steadily growing in importance, our second question is concerned with the implementation of synchronous models based on present-day platforms. Two aspects of synchronous program implementation will be briefly described. Firstly, we will discuss translation to sequential programs. An important issue in this respect is the synthesis of multitasking implementations, and the correct choice and configuration of inter-process communication (IPC). Secondly, for distributed implementation, a loose synchronization scheme for event-triggered bus systems such as CAN is described.

Translation to sequential programs Singlethread sequential code generation for synchronous programs has been intensively studied [HRR91], and efficient code generation from synchronous models is already provided by numerous commercial code generators. A specific problem in this context, which has been addressed in the AutoMoDe project [BBR^{+}05], is the mapping of synchronous data-flow designs in AutoFOCUS to the ASCET tool of ETAS. ASCET models, which basically combine graphical dataflow modeling with sequential language semantics, are used as a suitable intermediate format. The availability of highly optimized, production-quality code generation for automotive embedded platforms makes ASCET especially interesting in this respect.

For the AutoFOCUS-ASCET translation, optimization issues like minimal allocation of heap variables for data-flow algorithms have been addressed. The resulting translation algorithm synthesizes optimized ASCET methods or processes from AutoFOCUS data-flow designs.

Translation to multitasking implementations A specific issue that has been dealt with in the AutoMoDe project [BBR^{+}05] is semantics-preserving inter-task communication for pre-emptive OSEK task sets using data consistency mechanisms[3]. In wait-free IPC, the operating system uses a double-buffering technique to ensure that a reader task will have consistent input values for the duration of its activation. Using wait-free IPC, the implementation scheme developed in AutoMoDe can implement a zero-delay or unit-delay communication link specified

[3] There are a number of inter-task communication primitives known from the literature which ensure atomic and consistent accesses to shared data. For control law computations in resource-restricted applications, wait-free IPC has emerged as the mechanism of choice [PMSB96].

in the synchronous AutoFOCUS model. The kind of implemented delay depends on the relation of the sender and receiver activation frequency.

Translation to distributed implementations In order to implement a synchronous design, distributed processes need to match received messages to time indices from a common time base. Present-day bus systems such as Controller Area Network (CAN) are not designed for providing a common, fault-tolerant time base. Furthermore, message latencies be locally determined. [RB04] presents an implementation scheme for synchronous programs based on event-triggered bus system such as CAN. The technique guarantees semantics-preserving implementation under given operational conditions. The implementation scheme is based on a simple, fault tolerant synchronization layer, which uses application data traffic to keep nodes loosely synchronized.

For time-triggered distributed architectures such as TTA or the static segment of FlexRay, the deterministic assumptions of the model are naturally reflected by the platform. [CCM+03] demonstrates how designs in the synchronous tool SCADE can be mapped to such architectures.

3 Conclusion, and open issues

The synchronous programming model provides a precise basis for specifying behavior and timing of distributable models. For embedded controller design, comfortable and understandable notations are available that support both discrete and continuous controller design. For predicting behavioral properties of the final implementation, the synchronous approach has some compelling advantages over other popular approaches, such as singlequeue UML-RT.

Despite the progress in implementation techniques, the inherent conflict of the strong timing and concurrency assumptions in synchronous models with industrial needs necessitates further research. One could envision "soft" conformance relations for synchronous designs, which allow to weaken some of the inherent assumptions in a well-defined manner for implementing systems. A first step in this direction are GALS (Globally Asynchronous, Locally Synchronous) techniques for synchronous designs, which are mainly suited for discrete controllers [Ben01], or related loosely coupled schemes such as [RB04]. Soft conformance techniques for continuous and mixed discrete/controllers would be an important contribution to the practical applicability of the synchronous approach.

References

[BBR+05] A. Bauer, M. Broy, J. Romberg, Bernhard Schätz, P. Braun, U. Freund, N. Mata, R. Sandner, and D. Ziegenbein. AutoMoDe: Notations, Methods, and Tools for

Model-Based Development of Automotive Software. In *Proceedings of the 2005 SAE World Congress*, Detroit, MI, 2005. Society of Automotive Engineers.

[BCE+03] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone. The Synchronous Languages Twelve Years Later. *Proceedings of the IEEE*, 2003.

[Ben01] A. Benveniste. Some synchronization issues when designing embedded systems from components. In *Proceedings of 2001 Workshop on Embedded Software, EMSOFT'01*, volume 2211 of *LNCS*, pages 32–49, 2001.

[CCM+03] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to SCADE/Lustre to TTA: A layered approach for distributed embedded applications. In *Proceedings of the 2003 ACM SIGPLAN conference on Languages, Compilers, and Tools for Embedded Systems*, pages 153–162. ACM Press, 2003.

[DS02] L.A.J. Dohmen and L.J. Somers. Experiences and Lessons Learned Using UML-RT to Develop Embedded Printer Software. In M. Oivo and S. Komi-Sirviö, editors, *Proceedings of PROFES 2002*, volume 2559 of *LNCS*, pages 475–484, 2002.

[Han03] Software Tools Heading Mainstream. *The Hansen Report on Automotive Electronics*, July/August 2003.

[HRR91] N. Halbwachs, P. Raymond, and C. Ratel. Generating Efficient Code from Data-Flow Programs. In *Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming*, Passau, Germany, 1991. Springer Verlag.

[OMG05] Object Management Group. *Unified Modeling Language (UML) Superstructure*, August 2005. Version 2.0 formal/05-07-04.

[PMSB96] S. Poledna, Th. Mocken, J. Schiemann, and Th. Beck. ERCOS: An Operating System for Automotive Applications. Research Report 21/1996, Technische Universität Wien, Institut für Technische Informatik, 1996.

[RB04] J. Romberg and A. Bauer. Loose Synchronization of Event-Triggered Networks for Distribution of Synchronous Programs. In *Proceedings 4th ACM International Conference on Embedded Software*, Pisa, Italy, September 2004.

[Sel03] B. Selic. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, 2003.

[SPFR98] M. Saksena, A. Ptak, P. Freedman, and P. Rodziewicz. Schedulability analysis for automated implementations of real-time object-oriented models. In *IEEE Real-Time Systems Symposium*, 1998.

[SRS+03] B. Schätz, J. Romberg, M. Strecker, O. Slotosch, and K. Spies. Modeling Embedded Software: State of the Art and Beyond. In *Proceedings of ICSSEA 2003 16th International Conference on Software and Systems Engineering and their Applications*, 2003.

Testing of Embedded Control Systems with Continous Signals

Ina Schieferdecker	Jürgen Großmann
Technical University Berlin	DaimlerChrysler AG
Fraunhofer FOKUS	
schieferdecker@fokus.fraunhofer.de	juergen.grossmann@daimlerchrysler.com

Abstract

The systematic testing approaches developed within the telecommunication domain for conformance and interoperability testing of communication protocols have been extended and broadened to allow the testing of local and distributed, reactive and proactive systems in further domains such as Internet, IT, control systems in automotive, railways, avionics and alike. With the application of these testing principles it became apparent that the testing of systems with continuous systems is different to that of discrete systems. Although every continuous signal can be discretized by sampling methods (and hence mapped to the discrete signal paradigm), abstraction and performance issues in this setting become critical. This paper investigates the different options to support tests of embedded control systems in the framework of TTCN-3.

1 Introduction

Control systems in real-time and safety-critical environments are hybrid: they use both discrete signals for communication and coordination between system components and continuous signals for monitoring and controlling their environment via sensor and actuators. A test technology for control systems needs to be able to analyze the functional and timed behaviour of these systems being characterized by a set of discrete and continuous signals and their relations. While the testing of discrete controls is well understood and available via the standardized test technology TTCN-3[9], the specification-based testing of continuous control and of the relation between the discrete and the continuous control is not. A variety of test methods exist, which address different aspects – however, the concepts have not yet been consolidated. This is also shown by the fact that consortia like AutoSar, ESA or the MOST Forum are looking for an appropriate test technology. Appropriateness of a test technology is to be understood from different perspectives:

1. The adequateness of testing concepts and their syntactical presentation (textual or graphical) to the user
2. The applicability to various elements of the control systems, e.g. to discrete, continuous, local and distributed components
3. The applicability to various target systems, e.g. application components, communication components, control components, etc.

4. The portability to various target platforms, e.g. to platforms of the component providers and component integrators as well as to platforms of the test device vendors
5. The usability for different development approaches, e.g. HIL, SIL, MIL, etc.
6. The usability for different testing kinds, e.g. component level tests, system level tests, acceptance level tests, functional and non-functional tests, regression tests

TTCN-3 as such addresses already the majority of these aspects: it has constructs for

- message-based and procedure-based communication (making it applicable for asynchronous, decoupled and synchronous systems in client-server or peer-to-peer architectures),
- detailed test behaviours which allow to describe sequential, alternative, looping or parallel testing scenarios
- local and distributed test setups, so that physically distributed and/or heterogeneous systems with different access technologies can be tested
- an open and adaptable architecture to customize an executable test system to various target systems and target test platforms
- different presentation formats to highlight specific aspects of the test definitions or make it better fit to an application domain
- accessing different data formats of the system/component to be tested

It is a platform-independent, universal and powerful test specification technology, which allows selecting the abstraction level for the test specifications – from very abstract to very detailed, technical test specifications. It is also a test implementation language, which precisely define how to convert the abstract test specifications into executable tests. Also the tracing and evaluation of test results allows different presentation formats for textual and graphical logs. However, TTCN-3 lacks continuous signals as a language construct – which limits the power of TTCN-3 with respect to the adequateness of test concepts, the applicability to various target systems and the usability for different development approaches in the context of control systems. This paper discusses various options for testing continuous controls and analyses the use/extension of TTCN-3 for testing automotive control systems, in particular.

The paper is structured as follows: Section 2 reviews current approaches of testing control systems in the automotive domain. Section 3 discusses the basic concepts for testing continuous controls. Section 4 analyses how TTCN-3 can be used and extended to address the testing of continuous controls. Section 5 presents an example. Conclusions finish the paper.

2 Related Work

Established test tools in automotive from e.g. dSPACE [2], Vector[3] etc. are highly specialised for the automotive domain and come usually together with a test scripting approach which directly integrates with the respective test device. However, all these test definitions special to the test device and by that not portable to other platforms and not exchangeable. Some, e.g. ETAS [4], have already adopted TTCN-3, however, have taken it as it is without analysing further if the specific needs of the automotive domain are better to be reflected in the language itself. Hence, other platform independent approaches have been developed such as CTE [5], MTEST [6], and TPT [1]. CTE supports the classification tree method for high level test designs, which are applicable to partition tests according to structural or data-oriented differences of the system to be tested. Because of its ease of use, graphical presentation of the test structure and the ability to generate all possible combination of tests, it is widely used in the automotive domain. However, the detailed specification of test procedures and the direct generation of executable tests are out of consideration. MTEST is an extension of CTE to enable also the definition of sequences of test steps in combination with the signal flows and their changes along the test. By that it adds continuous signals to the test description, has however only limited means to express test behaviours which go beyond simple sequences, but are typical for control systems. This is addressed by TPT, which uses an automaton based approach to model the test behaviour and associate with the states pre- and post-conditions on the properties of the tested system (incl. the continuous signals) and on the timing. In addition, a dedicated run-time environment enables the execution of the tests. TPT is a dedicated test technology for embedded systems controlled by and acting on continuous signals, however, the combination with discrete control aspects is out of consideration. Therefore, this paper discusses first ideas on how to combine TPT concepts with TTCN-3 so as to gain the most from both approaches. It has been decided to adopt TPT concepts within TTCN-3 in order to make also advantage of TTCN-3 being a standardized test technology and having the potential to included dedicated concepts for continuous signals in later versions of TTCN-3.

3 TPT Concepts

TPT [1] uses for the modelling of test cases state machines combined with stream processing functions [11]. Every state describes a specific time quantified phase of a test. The transitions between states describe possible moves from one phase into the other. Every path through the automaton describes a possible execution of the test. The behaviour within the phases is formalized by expressions on the input and output flows of the channels the test is using. As these expressions are formal and not easy to understand, the graphical automaton uses informal annotations to the transitions and phases which explain the purpose, but do not give the precise definition which is hidden in the formal definitions. Semantically, an execution of such a hybrid system is a sequence of states and transitions, where the states describe the behaviour of the system up until the next transition is firing. This is reflected by the semantic concept of a component, which receives inputs via input channels and produce outputs via output channels. A *channel* is a named variable, to which streams can be assigned. A *stream* s of type T is a total function s: $\mathbb{R} \rightarrow T \cup \varepsilon$, where $\mathbb{R}$ denote real numbers and represent a dense time domain, and ε[1] represents the absence of a value, meaning that for s(t) = ε that there is no value of s at time t. Let C be a finite set of channels. Every channel $\alpha \in C$ has a type T_α. An *assignment* c of the channels C assigns to every channel $\alpha \in C$ a stream c_α: $\mathbb{R} \rightarrow T_\alpha \cup \varepsilon$. $\vec{C}$ is the set of all possible assignments of C.

A *component* is a relation F: $\vec{I} \leftrightarrow \vec{O}$, which defines the assignments of output channels $\vec{O}$ in dependence of the assignments of input channels $\vec{I}$. This relation has to have the time causality property: it exists a δ>0, so that for all i, i' ∈ $\vec{I}$ and o, o' ∈ $\vec{O}$ with o $=_{<0}$ o', (i,o) ∈ F and (i',o') ∈ F and for all $t \geq 0$ the following holds:

i $=_{\leq t}$ i' and o $=_{\leq t-\delta}$ o' implies that o $=_{\leq t}$ o'

Please note that c $=_{<t}$ c' and c $=_{\leq t}$ c' denotes the fact, that the assignments o and o' are equal in the time interval $(-\infty,t)$ and $(-\infty,t]$, resp.

TPT is based on the following model of hybrid systems: H = (V, E, src, dest, init, I, O, behaviour, cond) with

- dest, init) being a finite automaton having a finite set of states V, a finite set of transitions E, a function src: E → V, which assigns to every transition the source state, a function dest: E → V, which assigns to every transition the target state, a special initial state init ∈ V
- (I, O) being a signature with a finite set of input channels I, a finite set of output channels O and $I \cap O = \emptyset$, a function behaviour: V → $\vec{I} \leftrightarrow \vec{O}$ which assigns to every state v ∈ V a component $\vec{I} \leftrightarrow \vec{O}$, which is called behaviour, and a function cond: E → ($\overrightarrow{I \cup O} \rightarrow \mathbb{R} \rightarrow \mathbb{B}$), assigning to every transition e ∈ E a condition cond ∈ $\overrightarrow{I \cup O} \rightarrow \mathbb{R} \rightarrow \mathbb{B}$ for which the inputs O are delayed effective and where $\mathbb{B}$ are the Boolean values. The transition fires at the earliest (V, E, src, time when cond is true.

A component defines the values of (hybrid H): $\vec{I} \leftrightarrow \vec{O}$ for which holds: Let i ∈ $\vec{I}$ and o ∈ $\vec{O}$. Then is (hybrid H)(i,o) iff there is a finite or infinite sequence of states v ∈ V: φ= $v_1 \xrightarrow{e1} v_2 \xrightarrow{e1} \ldots$ where

- v_1= init and $src(e_n)=v_n$ and $dest(e_n)=v_{n+1}$, i.e. it is a correct path of the automaton H
- the concatenation behaviour(e1) –cond(e1)-> behaviour(e2) … is a sequentialation of H, and
- every other path φ ' which fulfils the above two properties is "slower" than φ

[1] ε will later be mapped to omit in TTCN-3.

Please refer to [1] for the complete definitions as only an outline of the basic semantics of TPT can be given here.

4 TTCN-3 for Hybrid Systems with Discrete and Continuous Signals

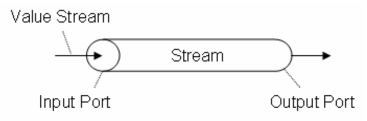
In order to enable the definition of tests for hybrid systems based on the semantic model of TPT, TTCN-3 needs to be extended with

- a concept of streams and channels (input and output channels and local channels),
- a concept of continuous time and sampling rates, and
- a concept of assignments to/evaluations of channels by direct definitions and by time partitions.

4.1 Streams and stream ports

The concept of channels is mapped to the concept of ports by adding an additional port kind for stream-based ports:

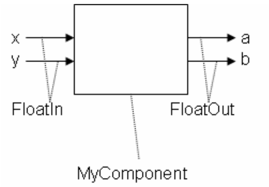
```
type port FloatIn stream {

       in float
}
type port FloatOut stream {
       out float
}
```



Stream ports are allowed to have an in or an out direction only, but not both (and also not an inout definition). Although TPT proposes to use float, integer and boolean for streams only, we leave it open to use any type, i.e. also user defined, structured type, as the basis of a stream-based port.

A test component having three input and output channels is then defined by a normal TTCN-3 test component type:

```
type component MyComponent {
       port FloatOut a,b;
       port FloatIn x,y
}
```



These leaves also the possibility open to combine continuous and discrete (i.e. message- or procedure-based) ports as the interface of a component type. In order to use streams for internal calculations (like TPT is using local channels), streams are defined as separate language concept:

```
stream float s;
```

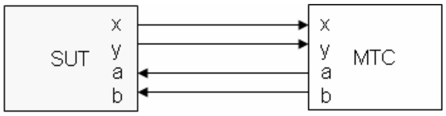
4.2 Evaluation of input streams

The essential concept is to find a way to define the generation of an output stream and the analysis of an input stream. For discrete ports, TTCN-3 uses combinations of send/receive, call/getcall, reply/getreply and raise/catch for the generation of outputs and analysis of inputs. The direct definition of a test uses a set of equations, which define how the local and output channels behave in dependence of the input channels: the set consists of equations $g_{c_1} ; g_{c_2} ; \ldots ; g_{c_n}$, where for each channel $c_i \in O \cup L$ an equation in the form $c_i(t) = E_i$ has to exist. The expressions E can use channels from $I \cup O \cup L$, where it is required that there is no cyclic dependency between the channels, but a delayed effectiveness, i.e. the set of equations can be resolved directly by term evaluation. This condition holds if for every $t \in \mathbb{R}$, the values of the output channels are defined by use of the values of the input channels for $t' \leq t$ and by use of the values of the output channels for $t'<t$ only. An example for such an equation system is given in the following:

```
x(t) = (t < 2) ? a(t) : x(t-2)
l(t) = b(t) + 2
y(t) = l(t/2) * sin(2 * t)
```

where l is a local stream.

Conceptually, we use this equation system as a characterization of the system under test (SUT), i.e. the input streams are to be interpreted as the inputs to the SUT which is required to react with outputs along the definition of the output stream:



This is reflected with the following testcase header

```
testcase MyTestCase() // a specific testcase
runs on MyComponent   // running on the main test component
system MyComponent    // testing a system of MyComponent
{ … }
```

Please note that the interpretation of the test system interface (defined with the system clause) is interpreted from the test system perspective, i.e. a and b are outputs from the test system and finally inputs to the system under test. x and y are the outputs from the SUT and inputs to the test system and being analyzed there.

The matching for the stream uses the equation system as proposed by TPT. The expressions in TPT use elements which are also available in TTCN-3. i.e. values, constants, type conversion (in contrast to TPT, TTCN-3 uses explicit conversion functions and not type casts), arithmetical operators and boolean operators. The conditional expression (_)?_:_ operator should be added to TTCN-3 in order to ease the definition of the expressions.

```
function Eval_DependentStream() runs on MyComponent {
        stream float l;       // local stream
        sample t(0.2); // sample rate for evaluation
        x@t == (t < 2) ? a@t : x@(t-2);
        l@t == b@t + 2 ;
        y@t == l@(t/2) * sin(2 * t) ;
}
```

For that, the @- and the ==-operator are introduced on stream ports: the @-operator represents the value of the stream at time t, which is taken from the sample rate defined in the sample statement, i.e. streams are discretized as defined by TPT for their evaluation. The ==-operator is the operator to define the equation system for the stream characterization. Whether evaluation is performed by an offline or online analysis depends on the test system realization and is currently outside the scope of consideration. Potentially, it could be useful to make it explicit in the test specification to require an online or offline evaluation, but this is for further study. In addition, a #-operator is used to get access to the value, which was in the stream before the current evaluation with the @-operator. Hence, x#(t+0.2) equals x@t as 0.2 is the sampling rate of t. Please note, that TPT uses for the @-operator a dot notation "." and for the #-operator a "-@" notation. The "." notation was not chosen here, as the dot has already a different meaning in TTCN-3. The "-@" notation was not chosen in order to have single sign operators only.

Another case is a set of streams that are independent from other streams, so that the equation is defined as an expression without referring to other streams:

```
function Eval_IndependentStream() runs on MyComponent {
        sample t(0.2); // sample rate for evaluation
        x@t == 2 * t - 3;
}
```

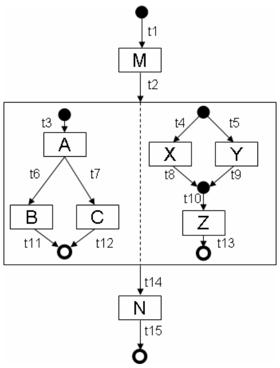
4.3 Generation of output streams

In the case that the test system has to provide streams as inputs to the SUT only, the equation systems as introduced above can be used. However, this time the equations are independent from other streams, i.e. the stream is defined by an equation and generated. Hence, the assignment-operator ":=" is used instead of the evaluation-operator "==". Both equation systems presented in Section 4.2 can e.g. be used (with an assignment operator and with output channels):

```
function Generate_IndependentStream() runs on MyComponent {
        sample t(0.2); // sample rate for generation
        a@t := 2 * t - 3;
}

function Generate_DependentStream() runs on MyComponent {
        stream float l;        // local stream
        sample t(0.2); // sample rate for evaluation
        a@t := (t < 2) ? x@t : a@(t-2);
        l@t := y@t + 2 ;
        b@t := l@(t/2) * sin(2 * t) ;
}
```

4.4 System characterization by time partitions

Test cases with time partitions use the model of hybrid systems introduced in Section 0. TPT state machines are represented by initial and final states, states with associated functions as defined in Section 4.2 and 4.3, junctions and parallelization of behaviours. The test components in TTCN-3 have sequential behaviours. Several test components are used for parallel behaviours. Control structures with conditionals and loops can be used to represent the junctions. The central question is how to combine evaluation and generation functions in the states with the control flow of the state machines. An example TPT definition is given on the right hand side.



We assume that every transition t has a condition c_t and for every state there is a function characterizing the continuous behaviour in that state as defined in Section 4.2 and 4.3. We propose two new TTCN-3 statements:

- The **try**-statement to check the fulfilment of the transition preconditions. It can check several conditions and is successful if one of the conditions evaluates to true at the current point in time.

- The **carry**-**until**-statement to perform a behaviour (a continuous, discrete or hybrid one) up until the given condition becomes true.

The evaluation of the condition is done according to the current sample rate. These two statements allow to define junctions/states with several enabling conditions (such as t4 and t5) and states with several outgoing transitions (such as A) although this may require the double check of a condition (such as to enable the right hand parallel part and to select the X or the Y branch):

```
testcase TPT_TestCase()
runs on MyComponent system MyComponent {
        sample t(0.2);
        try {
        [] c_t1 {
                carry { M() } until c_t2;
                carry { FPar() } until c_t14;
                carry { N() } until c_15 ;
        }
}

function FPar() runs on MyComponent {
        var MyComponent l,r;
        l:= MyComponent.create;
        r:= MyComponent.create;
        l.start(FPar_l());
        r.start(FPar_r());
}
function FPar_l() runs on MyComponent {
        try {
        [] c_t3 {
                carry { A() } until (c_t6 or c_t7);
                try {
                [] c_t6 { carry { B() until c_t11: }
                [] c_t7 { carry { C() until c_t12; }
                }
        }
}
function FPar_r() runs on MyComponent {
        try {
        [] c_t4 {
                carry { X() } until (c_t8);
           }
        [] c_t5 {
                carry { Y() } until (c_t9);
           }
        }
        try {
        [] c_t 10 {
                carry { Z() } until (c_t13);
           }
        }
}
```

The try and carry-until statements assure that all control cases being proposed by TPT can be represented. We will further analyse the usage of these statements also in combination with the other TTCN-3 statements.

5 Summary

This paper reviews the requirements for a test technology for embedded systems, which use both discrete signals (i.e. asynchronous message-based or synchronous procedure-based ones) and continuous flows (i.e. streams). It compares the requirements with the only standardized test specification and implementation language TTCN-3 (the Testing and Test Control Notation [9]). While TTCN-3 offers the majority of test concepts, it has limitations for testing the aspects of continuous streams. Therefore, the paper reviews current approaches in testing embedded systems and identifies commonalities between TTCN-3 and TPT (the Time-Partition-Test method [1]) in terms of abstractness and platform-independence.

Subsequently, the paper investigates ways to combine TPT concepts with TTCN-3, so as to preserve the standard base of TTCN-3 and extend it to continuous signals. For that, TTCN-3 is being extended with concepts of streams, stream-based ports, sampling, equation systems for continuous behaviours and additional statements that allow control flows of continuous behaviours. The paper demonstrates the feasibility of the approach by providing a small artificial example. Future work will further refine the inclusion of TPT concepts into TTCN-3 and consider e.g. the combination of discrete and continuous control within one test behaviour function on a test component instance having both discrete and continuous ports. The concepts will be implemented in a TTCN-3 tool set [12] and applied to a real case study in the context of CAN-based engine controls in a car.

6 References

[1] E. Lehmann: Time Partition Testing, Systematischer Test des kontinuierlichen Verhaltens von eingebetteten Systemen, Promotion, Technischen Universität Berlin, November 2003.

[2] ControlDesk Test Automation Guide For ControlDesk Version 1.2. dSPACE GmbH, Paderborn (D), Sept. 1999

[3] B. Tettenborn, A, Leuze, S. Meißner: PXI basierendes Funktionstestsystem für Motorsteuergeräte im Rennsport, White Paper, May 2004.

[4] F. Tränkle: Testing Automotive Software with TTCN-3, 2nd TTCN-3 User Conference, June 6-8, 2005, Sophia Antipolis, France.

[5] M. Grochtmann, K. Grimm, J. Wegener: Tool-Supported Test Case Design for Black-Box Testing by Means of the Classification-Tree Editor. Proc. of 1. European Int. Conf. on Software Testing, Analysis and Re-view (EuroSTAR '93), London (GB), S. 169-176, Oct. 1993

[6] M. Conrad. Modell-basierter Test eingebetteter Software im Automobil: Auswahl und Beschreibung von Testszenarien. Dissertation, Deutscher Universitätsverlag, Wiesbaden (D), 2004

[7] K. Lamberg, M. Beine, M. Eschmann, R. Otterbach, M. Conrad, I. Fey: Model-based Testing of Embedded Automotive Software using MTest. SAE World Congress 2004, Detroit (US), März 2004

[8] ETSI : TTCN-3 Homepage, http://www.ttcn-3.org, June 2005.

[9] ETSI ES 201 873-1 V3.1.1, Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language, Sophia Antipolis, France, July 2005.

[10] J. Zander, Z.R. Dai, I. Schieferdecker, G. Din: From U2TP Models to Executable Tests with TTCN-3 - An Approach to Model Driven Testing, TestCom'05, Canada, Montreal.

[11] M. Broy. Refinement of Time. In Transformation-Based Reactive System Development, LNCS 1231, Springer-Verlag, 1997.

[12] Testing Technologies IST GmbH: TTworkbench v1.3, http://www.testingtech.de, Oct. 2005.

Domänenspezifische Integration von Modellierungswerkzeugen mit Sichten

Andy Schürr, Johannes Jakob

FG Echtzeitsysteme
TU Darmstadt
Merckstr. 25
64283 Darmstadt
andy.schuerr@es.tu-darmstadt.de
johannes.jakob@es.tu-darmstadt.de

Zusammenfassung: Modellbasierte Softwareentwicklung setzt zum einen voraus, dass die mit verschiedenen Werkzeugen erstellten Entwicklungsartefakte (Modelle) durch Konsistenzprüfungen und Transformationen integriert werden; andererseits ist eine Anpassung vorhandener Werkzeuge und ihrer (Meta-)Modelle an die Spezifika einer Domäne eine oft genannte Forderung. Nur so können durchgängige Entwicklungsprozesse mit domänenspezifischen Modellierungsrichtlinien erfolgreich operationalisiert werden. Kann man für die Anpassung der verwendeten Modellierungskonzepte an eine Domäne nicht auf Meta-CASE-Technologie zurückgreifen, so bleibt oft nur die Definition von Sichten (ad-hoc-Erweiterungen) auf den Schnittstellen der verwendeten Werkzeuge. In diesem Beitrag befassen wir uns deshalb mit der Kombination von Konzepten zur Spezifikation von Modellsichten und Modelltransformationen, eine bislang in der SE-Gemeinde weitgehend vernachlässigte Thematik

1. Einleitung

Die Integration von (Software-)Entwicklungs- bzw. Modellierungswerkzeugen zur besseren Unterstützung durchgängiger Entwicklungsprozesse ist seit nunmehr fast 30 Jahren ein wichtiges Forschungsthema der Software-Engineering-Gemeinde. Auf technischer Ebene wurden in den vergangenen Jahrzehnten Lösungen zur Benutzeroberflächenintegration mit Hilfe von GUI-Toolkits, zur Datenintegration über gemeinsame Repositories bzw. Datenbanken sowie zur Kontrollintegration vermittels ereignisbasierter Mechanismen entwickelt [2]. Zur Datenintegration auf einer semantisch höherstehenden Ebene haben sich in jüngster Zeit zudem *Modelltransformationsansätze* mit entsprechenden Sprachen und Werkzeugen etabliert [6]. Neben vielen proprietären Ansätzen wie ATL, AMW, ArcStyler, ATOM3, BOC, GreAT, OptimalJ, Tefkat, etc. ist hier besonders der jüngst verabschiedete OMG-Standard QVT zu nennen (siehe hierzu auch [1, 10, 12, 14]).

Das Akronym „QVT“ steht für „*Query, View and Transformation*“; die gerade verabschiedete Version 2.0 der RFP-Submission [10] deckt aber von den drei genannten Aspekten nur die Beschreibung von Anfragen auf Modellen und die Spezifikation von Transformationen auf verschiedenen Abstraktionsniveaus ab. Die *Definition von Sichten* auf Modellen wird bereits auf den ersten Seiten des Standards ausgeklammert. Selbiges gilt für alle anderen uns bekannten Modelltransformationsansätze, insbesondere wenn man die Anforderung nach so genannten „updatable Views“ erhebt. In der Datenbankwelt sind hingegen Sichten, auf denen auch schreibende Zugriffe unterstützt werden (Updates), seit vielen Jahren das wichtigste Abstraktionsmittel, um Datenbankanwendungen vor Änderungen der von Ihnen benutzten Datenbanken zu schützen.

Es gibt mindestens vier wichtige Gründe, die für die *Einführung von Sichten* auf Modellen und Werkzeugdaten sowie für die Durchführung von Transformationen auf solchen Sichten sprechen:

1. Traditionell werden Sichten im Software-Engineering dafür eingesetzt, einen Sachverhalt aus verschiedenen Perspektiven zu betrachten und zu beschreiben; sie bilden damit benötigte *Projektionen* auf *einen* integrierten Datenbestand und unterstützen u.a. die rollenspezifische Filterung und Vergabe von Zugriffsrechten auf Entwicklungsdaten.
2. Des Weiteren werden Sichten dafür eingesetzt, von den Schnittstellen einzelner Werkzeuge für denselben Anwendungsbereich zu abstrahieren und damit die *Austauschbarkeit* des Werkzeugs eines Herstellers durch das eines anderen zu gewährleisten (soweit Standards nicht hierfür zur Verfügung stehen).
3. Bei der Spezifikation von Transformationen zwischen sich strukturell stark unterscheidenden Modellen hat es sich auch als zweckdienlich erwiesen, zunächst Sichten zur *Angleichung* der betrachteten Modelle einzuführen und anschließend die benötigte Transformation auf diesen Sichten zu beschreiben.
4. Schließlich eignen sich Sichten dafür, zunächst genau auf eine *Anwendungsdomäne* (oder gar ein Projekt) zugeschnittene Modelle (z.B. für „Automotive Software Development“) einzuführen und diese anschließend als Sichten auf allgemeiner verwendbaren Werkzeugdatenstrukturen zu implementieren.

Deshalb sind Modelltransformationsansätze so zu erweitern, dass sie die Definition von Modellsichten *effizient* unterstützen; dabei sind sowohl lesende als auch schreibende Zugriffe über einer möglichst großen Klasse von Sichtdefinitionen zu erlauben.

Im Folgenden werden wir einen Ansatz skizzieren, der *einen formalen Apparat*, so genannte Tripelgraphgrammatiken, für die Definition von *Modelltransformationen und Modellsichten* verwendet. Im nachfolgenden Abschnitt 2 wird hierfür ein (abstraktes) Beispielszenario skizziert, das den kombinierten Einsatz von Modellsichten und Transformatoren bzw. -integratoren motiviert. Anhand dieses Beispiels werden Anforderungen an einen Ansatz zur Spezifikation und Implementierung von Modellsichten aufgestellt. In Abschnitt 3 werden dann bekannte Datenbank- und Modelltransformations-Ansätze darauf hin untersucht, ob die ihnen zugrunde liegenden Konzepte für die Definition von Modellsichten einsetzbar sind. In Abschnitt 4 wird schließlich das von uns verfolgte Konzept zur Definition von Modellsichten skizziert, ohne dort im Rahmen eines Positionspapiers auf technische Details eingehen zu können.

2. Beispiel für kombinierte Modellsichten und –transformationen

Bei der Entwicklung eingebetteter (sicherheitskritischer) Softwaresysteme hat sich in den letzten Jahren mehr und mehr eine modellbasierte Vorgehensweise etabliert, die die informelle Beschreibung von Anforderungen als Textblöcke mit vorgegebener Struktur mit der (semi-)formalen Spezifikation ausführbarer Modelle kombiniert. Hinzu kommen Entwicklungsdaten für die Beschreibung auszuführender Tests, die Verteilung von Softwarekomponenten auf eine Hardwareplattform, Test- und Inspektionsprotokoll, … . Oft wird dabei eine Kombination aus einem Requirements-Engineering-Werkzeug wie DOORS oder RequisitePro und einem block- und/oder objektorientierten Modellierungswerkzeug wie Matlab/Simulink oder Rose/RT eingesetzt. Mit zunehmender Erfahrung im Einsatz solcher Werkzeuge für eine modellbasierte Vorgehensweise etablieren sich mehr und mehr auch Richtlinien, die auf einen bestimmten Anwendungsbereich zugeschnitten sind und beispielsweise vorgeben,

- wie textuelle Anforderungsdefinitionen zu strukturieren und zu „verlinken" sind (domänenspezifisches Metamodell für die Strukturierung von Anforderungen)
- wie blockorientierte Konzepte zur Definition von Software- und Hardwarearchitekturen einzusetzen sind (ADL-Metamodelle) und
- wie konsistente Paare von Anforderungsdefinitionen und Software-/Hardware-Architekturmodellen aussehen (durch Integrations- bzw. Transformationsregeln zwischen den oben genannten Metamodellen).

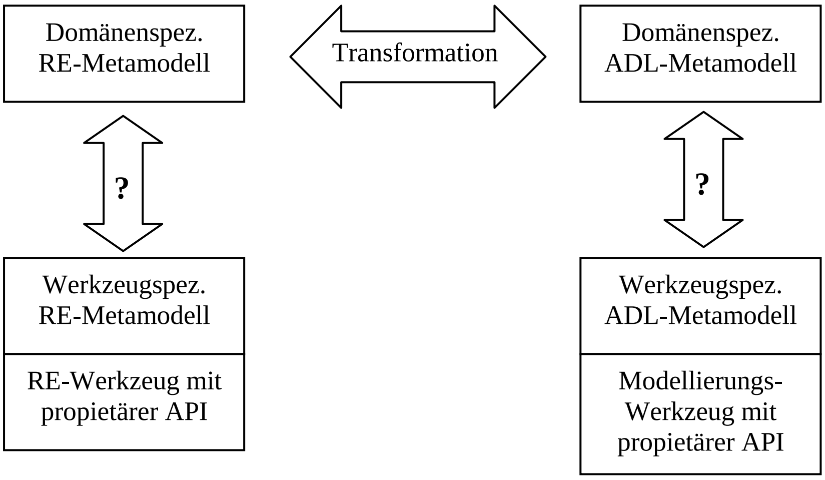


Abb. 1: Domänenspezifische Anpassung/Integration v. Modellierungswerkzeugen

Damit kommt man zu dem in Abb. 1 skizzierten Szenario. Für die beiden Aufgabenbereiche „Requirements Engineering" und „Architekturmodellierung" gibt es jeweils domänenspezifische Konzepte, die in Form von Metamodellen (Klassendiagrammen, Datenbankschemata) festgelegt werden. Constraint-Sprachen wie OCL können darüber hinaus zur Definition von lokalen statischen Modellierungsrichtlinien eingesetzt werden.

Für die Beschreibung bekannter Abbildungsvorschriften zwischen den betrachteten Metamodellen lassen sich entweder wiederum Constraint-Sprachen wie xlinkit [9] oder OCL bzw. Modelltransformationssprachen wie QVT einsetzen. Unklar ist jedoch bislang, wie man die in Abb. 1 dargestellte vertikale *Sichtenbildung* formal spezifiziert und daraus effiziente Implementierungen ableitet. Für ein Konzept zur Definition von Modellsichten ergeben sich bereits aus dem geschilderten Szenario die folgenden Anforderungen:

1. Modellsichten müssen sowohl lesende als auch schreibende Zugriffe (für die auf ihnen durchgeführten Konsistenzprüfungs- und Transformationsprozesse) unterstützen.
2. Schreibende Zugriffe auf Modelle erfolgen i.a. sowohl über die zugehörigen Sichten als auch direkt über die Schnittstellen der zugrunde liegenden Modelle; deshalb sind Funktionen anzubieten, die die Konformität von Modellen (Modellteilen) mit den darauf definierten Sichten überprüfen.
3. Es ist eine n-stufige Sichtenbildung zu unterstützen, um so z.B. zwischen den domänenspezifischen RE-Modellen und den werkzeugspezifischen RE-Modellen ein generisches RE-Modell einfügen zu können.
4. Modellsichten sind als funktionale Zugriffsschichten auf die APIs der zugrunde liegenden Modelle zu realisieren (und nicht als materialisierte Sichten, die alle sichtbaren Daten des zugrunde liegenden Modells kopieren).
5. Manche Entwicklungswerkzeuge (wie z.B. DOORS oder Rose/RT) besitzen erweiterbare Metamodelle; ihre zugehörigen Sichten müssen deshalb reflektive APIs für den Umgang mit Metamodellerweiterungen anbieten.

2. Verwandte Arbeiten

Um Modellsichten im obigen Sinne zu spezifizieren, stehen uns mindestens zwei grundsätzlich verschiedene Vorgehensweisen zur Auswahl. Zum einen ist es nahe liegend, wohlbekannte *Techniken aus dem Datenbankbereich* mit Modelltransformationsansätzen zu integrieren. So unterstützen relationale Datenbanksysteme wie Oracle die Definition von Sichten als Ergebnisse von zunächst beliebigen Anfragen. Allerdings werden an Sichtendefinitionen mit Update-Möglichkeiten sehr viel härtere Anforderungen gestellt; meist wird nur die Projektion und Selektion von Spalten und Zeilen aus einer Basistabelle unterstützt. Zudem wird oft davon ausgegangen, dass die Sichten in materialisierter Form vorliegen und nicht als rein funktionale Schichten unter Einsatz bewährter softwaretechnischer Konzepte wie Adapter-Design-Pattern realisiert werden. Ähnliches gilt für alle Ansätze, die in die Kategorie „Data Warehouses“ fallen. Auch hier werden Daten aus verschiedenen Quellen konzentriert und in einer Datenbank materialisiert. Zudem liegt bei „Data Warehouses“ und anderen Konzepten zur Integration verteilter Datenhaltungssysteme mit so genannten Mediatoren der Schwerpunkt auf der Durchführung verteilter Anfragen über global und/oder lokal definierten Schemata. Die Durchführung von verteilten Updates wird meist nicht betrachtet. Ein bekannter Vertreter dieser Art ist AMOS II [11].

Etwas bessere Unterstützung für die Deklaration von Update-Operationen auf Sichten bieten Datenbanksysteme, die so genannte *„instead-of-Trigger“* kennen; sie übersetzen schreibende Zugriffe auf eine Sicht in Aufrufe der von einem DB-Entwickler bereitgestellter Update-Prozeduren. Es liegt dabei in der Verantwortung des Entwicklers, dass die für lesende Zugriffe einer Sicht definierenden Anfragen und die für schreibende Zugriffe benötigten Update-Prozeduren zueinander konsistent sind. Objektorientierte Systeme wie SBQL gehen hier einen Schritt weiter, in dem die verschiedenen Teile einer Viewdeklaration (Operationen für Objekterzeugung, Anfragen, Updates und Löschen) zu einer Viewdeklaration zusammengefasst werden [5].

Noch weiter gehen in jüngster Zeit entwickelte föderierte Datenbankansätze, die durch Einbeziehung von P2P-Technologien Anfragen und Updates auf Verbunden weitgehend autonom agierender Datenquellen unterstützen. So findet man etwa in [7, 8] ein Konzept beschrieben, das durch Schematransformationen die Übersetzung der Daten eines Schemas in Daten eines anderen Schemas unterstützt. Diese *Schematransformationen* bestehen im wesentlichen aus einer Abfolge primitiver Operationen zum Hinzufügen und Entfernen von Typen (Tabellen) in einem Schema. So wird schrittweise beschrieben, wie die Daten eines Schemas in Daten eines anderen Schemas zu übersetzen sind. Unter nicht näher genannten Randbedingungen sind diese Schematransformationen automatisch invertierbar und damit bidirektional. Damit können lesende und schreibende Zugriffe auf einer Sicht in lesende bzw. schreibende Zugriffe auf die zugrunde liegenden Datenbanken übersetzt werden. Dieses Konzept des *„both-as-view“*-Ansatzes kommt unseren Anforderungen aus Abschnitt 2 bereits sehr nahe. Störend ist hierbei nur, dass die auch als Modelltransformationen interpretierbaren Schematransformationen auf einem sehr niedrigen operationalen Niveau zu spezifizieren sind und eine Integration mit den im Software-Engineering-Umfeld verwendeten Modelltransformationskonzepten noch nicht zur Verfügung steht. Des Weiteren bleibt unklar, unter welchen Randbedingungen Schematransformationen invertierbar sind und in welchem Umfang temporäre Datenstrukturen während der schrittweise durchgeführten Übersetzungsprozesse angelegt werden müssen (Effizienz des Ansatzes).

Somit scheint es sinnvoll, sich der am Anfang dieses Abschnitts angekündigten zweiten Möglichkeit zuzuwenden und Sichten auf Modelle mit Hilfe bereits existierender Modelltransformationssprachen zu beschreiben. Die dabei zum Einsatz kommenden Transformationen müssen für schreibenden und lesenden Zugriff auf Sichten bidirektional definiert sein und Änderungen inkrementell in beide Richtungen propagieren. Des Weiteren ist es wünschenswert, dass Sichten nicht als vollständige Kopien/Transformationen aller relevanten Daten des zugrunde liegenden Modells realisiert werden, sondern ihre Zugriffschnittstellen (APIs) direkt auf den APIs der zugrunde liegenden Modelle implementieren. Die eben genannten Anforderungen erfüllt keiner der uns bekannten Modelltransformationsansätze zur Gänze. Selbst deklarative Ansätze wie die von uns entwickelten Tripelgraphgrammatiken (TGGs) [3, 4] oder der OMG-Standard QVT [10] sind bislang allenfalls für die Definition vollständig materialisierter Sichten geeignet.

4. Tripelgraphgrammatiken für Modelltransformationen und Sichten

Aus den oben genannten Gründen sind wir zur Zeit dabei, eine neue Variante der *Tripelgraphgrammatiken* (TGGs) zu entwickeln, die nicht wie bislang bidirektionale Transformationen und Konsistenzprüfungen zwischen extensional definierten Modellen beschreibt, sondern für intensional definierte Modellsichten eingesetzt werden kann, die sowohl lesende als auch schreibende Zugriffe erlauben. Dabei werden TGGs so verwendet, dass jeweils ein Metamodellelement der zu definierenden Sicht (Objekt einer virtuellen Klasse, Link einer virtuellen Assoziation) auf ein beliebiges komplexes Objekt-/Link-Geflecht des zugrunde liegenden (Meta-)Modells abgebildet wird. Dieses (Meta-)Modell kann seinerseits wiederum durch eine weitere TGG als Sicht eines anderen (Meta-)Modells realisiert sein. Ein vollständig neu zu entwickelnder Ansatz zur Übersetzung deklarativer TGG-Regeln in operationale Graphtransformationen der Metamodellierungs-Umgebung Fujaba/MOFLON sowie die anschließende Übersetzung in JMI-kompatiblen Java-Code führt dabei zu folgender Lösung:

- Extensional und als Sichten intensional definierte Modelle werden über gleichartig aufgebauten Java-APIs manipuliert, die auf dem Industriestandard JMI basieren.
- JMI unterstützt dabei sowohl die Verwendung getypter Schnittstellen für feste Metamodelle als auch die Verwendung reflexiver, ungetypter Schnittstellen für erweiterbare Metamodelle.
- Für die Abbildung der Sichten-API auf die zugrunde liegende Modell-API werden die üblichen Adapter-Design-Pattern eingesetzt (geplant sind Adapter mit und ohne Proxy-Objekten).
- Der Java-Code für die Abbildung aller lesenden und schreibenden Methoden der Sichten-API auf die zugrunde liegende Modell-API wird aus TGGs mit „normalen“ Graphtransformationen als Zwischenschritt automatisch erzeugt.
- Einschränkungen bezüglich der Verwendung von Berechnungsvorschriften für Attribute (automatische Invertierbarkeit) können durch die manuelle Ergänzung der generierten normalen Graphtransformationen umgangen werden.
- Bei der Implementierung von Wrappern für propietäre Werkzeugschnittstellen sind entweder die getypten oder die ungetypten JMI-Schnittstellen zu implementieren, die jeweils anderen Schnittstellen werden generiert.

Nähere Details zu der Definition von Sichten mit TGGs und ihrer Übersetzung in Java-Code können in [13] nachgelesen werden. Eine ausführlichere Darstellung ist an dieser Stelle aus Platzgründen nicht möglich. Abschließend sei jedoch noch darauf hingewiesen, dass die von uns eingesetzten TGGs eng mit dem OMG-Standard QVT verwandt sind. Die deklarative und visuelle (relationale) Transformationsteilsprache von QVT kann nämlich weitgehend als eine konkrete Notation für TGGs angesehen werden. Deshalb lässt sich der hier zur Diskussion gestellte Beitrag zur Definition von Modellsichten für und mit Modelltransformationen auch als ein Beitrag zur *Erweiterung von QVT um Sichten* ansehen.

Literaturverweise

[1] K. Czarnecki, S. Helsen: *Classification of Model Transformation Approaches*. OOPSLA 2003 Workshop on Generative Techniques in the Context of MDA (2003).

[2] H. Dörr, A. Schürr: *Special Section on Tool Integration Application and Frameworks*, Int. Journal on Software Tools for Technology Transfer (STTT), Vol. 6-3, Springer Verlag, (2004), 183-255.

[3] A. Königs, A. Schürr: *MDI - a Rule-Based Multi-Document and Tool Integration Approach*, Special Section on Model-based Tool Integration in Journal of Software and Systems Modeling (SOSYM), Academic Press (2005). Note: accepted for publication.

[4] A. Königs, A. Schürr: *Tool Integration with Triple Graph Grammars - A Survey*, Proceedings of the SegraVis School on Foundations of Visual Modelling Techniques, in: Electronic Notes in Theoretical Computer Science, Academic Press (2005), Note: accepted for publication.

[5] H. Kozankiewicz, K. Subieta: *SBQL Views - Prototype of Updateable Views*. ADBIS 2004 (http://www.sigmod.org/dblp//db/conf/adbis/adbis2004l.html).

[6] H. Kühn, M. Murzek.: *Interoperability Issues in Metamodelling Platforms.* Proc. 1st Int. Conf. on Interoperability of Enterprise Software and Applications (2005).

[7] P. McBrien and A. Poulovassilis: *Defining Peer-to-Peer Data Integration using Both as View Rules*. Proc. Workshop on Databases, Information Systems and Peer-to-Peer Computing, at VLDB'03, 91-107.

[8] P. McBrien, A. Poulovassilis: *Data Integration by Bi-Directional Schema Transformation Rules*. ICDE 2003, 227-238.

[9] C. Nentwich, L. Capra, W. Emmerich, A. Finkelstein: *xlinkit: a Consistency Checking and Smart Link Generation Service*. ACM Transactions on Internet Technology, 2(2), 151-185.

[10] QVT Merge Group: *Revised Submission for MOF 2.0 Query, View, Transformation Request For Proposal (ad/2005-03-02)*, Version 2.0.

[11] T. Risch, V. Josifovski, T.Katchaounov: *Functional Data Integration in a Distributed Mediator System*. P. Gray, L. Kerschberg, P.King, A. Poulovassilis (eds.): Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data, Springer Verlag (2003).

[12] A. Schürr, H. Dörr (eds.): *Special Section on Model-based Tool Integration*; Int. Journal on Software and Systems Modeling (SOSYM), Vol. 4-2, Springer Verlag (2004), 109-170.

[13] A. Schürrr, J. Jakob: *View creation of meta models by using modified Triple Graph Grammars*. submitted for publication.

[14] S. Sendall, W. Kozaczynksi: *Model Transformation: The Heart and the Soul of Model-Driven Software Development*. Vol. 20, No. 5, IEEE Software (2003), 42-45.

Derivation of Executable Test Models from Embedded System Models Using Model Driven Architecture Artefacts - Automotive Domain -

Justyna Zander-Nowicka[1], Ina Schieferdecker[1,2], Tibor Farkas[1]

[1]Fraunhofer FOKUS, MOTION
Kaiserin-Augusta-Allee 31
10589 Berlin, Germany

[2]Technical University Berlin,
Faculty IV, Straße des 17. Juni 135
10623 Berlin, Germany

{zander-nowicka, schieferdecker, farkas}@fokus.fraunhofer.de

Abstract: The approach towards system engineering compliant to Model-Driven Architecture (MDA) implies an increased need for research on the automation of the model-based test generation. This applies especially to embedded real-time system development where safety critical requirements must be met by a system. The following paper presents a methodology to derive basic Simulink test models from Simulink system models so as to execute them in the same framework as the system model. The meta-models for Simulink and Test are explained and will enable automatic transformation in the future. The testing concepts of UML 2.0 Testing Profile (U2TP) have been partially adopted, however also enriched so as to deal with continuous functions, real-time constraints and to mirror Simulink-specific features.

1 Introduction

Embedded systems development needs modelling, simulation and analysis of dynamic behaviour. One of the tools enabling these functionalities is Simulink [MatML] - being often used in the automotive industry. It supports linear and nonlinear systems, modelled in continuous time, sampled time or a hybrid of the two. Simulation means the execution of the system model. It enables the prediction of the behaviour of the system from a set of parameters and initial conditions. Testing, on the other side is a process of identifying the completeness and quality of developed software. It demands the definitions of system under test (SUT), test objectives, environmental constraints and test criteria. In the early phases of the V-Model [VML97] testing means also execution of the test model. This leads to a conclusion that simulation provided by Simulink, may be involved in testing, however test definition and test artefacts (i.e. timer actions, clocks, actions assigning the verdict from the outside) must be additionally provided. The tool offers a set of model verification blocks which enable to state if the system is build right, according to the requirements. Test harness can be modelled only to some extent, thus the motivation of the following work is to extend Simulink with additional libraries, features and facilities to let the generated test models be executed. The aim is to provide automotive industry with an integrated environment for modelling and executing system as well as for modelling and executing tests in one common framework. Another motivation is the Model-Driven Architecture (MDA) and its artefacts which can be adopted for testing. As MDA gained much momentum in industry, the focus is to use these concepts so as to show that retrieving executable test instances from system model can be supported via either manual or even automatic test model generation.

The paper is divided into five sections. After the introduction, Section 2 is devoted to the Simulink, Simulink Test, Environment Test Directive and Test Directive meta-models which should be provided behind the tools. Section 3 provides an example of retrieving the executable tests, which is possible by applying some restrictions during modelling. In Section 4 work related to test generation from system models for embedded systems in automotive domain is considered. In Section 5, the results are depicted and conclusions are drawn. Finally, future work challenges are outlined.

2 Meta-models Behind the System and Test Designs

MDA prescribes a set of model artefacts to be used along system development, how those models may be prepared and their relationships [MDA03]. It is an approach to system development that separates the specification of functionality from the specification of the implementation of that functionality on a specific technology platform. Meta Object Facility (MOF) [MOF04] together with MDA has shown that meta-model based language definitions impose new ways of defining language semantics as a separation of syntax and semantic concept space, of integrating languages via a common meta-model base and of generating and deriving models from other models via model transformers [SD04].

This paper presents an approach to test embedded systems along the MDA-based paths. Simulink and Test Simulink meta-models are both defined as MOF models. Test Directive and Environment Test Directive meta-models are created in the same manner so as to enhance automatic test models generation in the future work. Those additional meta-models serve as sources of information about test requirements and safety-critical requirements, respectively. Details concerning the meta-models are explained in the next Section. Transformation rules defined for the example in this paper are applied only manually. They define relations between source and target meta-classes of given meta-models [ZDS^{+}05]. The whole approach is shown in Figure 1.

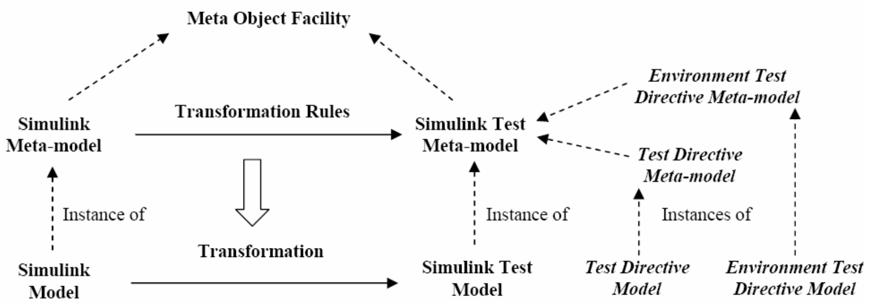


Figure 1: Retrieval of Simulink Test Models from Simulink Models Using the MDA Approach

Due to lack of explicit tool supporting MOF notation, the UML notation has been deliberately used to visualize selected meta-models.

2.1 Development of the Simulink Meta-Model

A proposal for Simulink meta-model (SL meta-model) has been developed by researchers from Vanderbilt University [NK04]. Although some insights of the core have been adopted, the following approach is slightly changed, extended and reflects more elements of the hierarchical, actor-oriented [Lee04] models. The meta-model defines the Simulink concept space with additional support for the graphical presentation format. It does not directly reflect the structure of Simulink models, but rather gives the semantics overview. It is defined in different packages with concept structures for Model, System, Block, Annotation, Line and Port, Parameter, Presentation, etc.

In this paper, only excerpts from the meta-model are presented because the main focus is put on the integration of system and test designs. The principal building blocks of system specifications in Simulink are models. Every model contains a single system. A system may also contain blocks that are used for modelling the behaviour, lines for blocks' connection or annotations used to add user's information. Some of these blocks may be typed by the subsystem. A subsystem can be built in the same way as a block. Every subsystem contains a single system [NK04], which enables hierarchical representation of complex embedded systems.

2.2 Development of the Simulink Test Meta-Model

The Simulink Test meta-model (SLTest meta-model) is developed from scratch, however the experience gained from UML 2.0 Testing Profile [U2TP04] has been used. It defines the testing concept space with special attention put on the Simulink-specific features. It does not directly reflect the structure of Simulink Test models, but rather gives the semantics overview. It is provided in different packages with concept structures for Test Architecture – describing test architecture, Test Behaviour – defining specific aspects of the test behaviour, Clock for Test – dealing with time constraints, test case timer actions and clocks, finally Simulation Test Data – for supplying continuous test data. It re-uses also some concepts provided within the SL meta-model (i.e. Model meta-class or blocks of type Model Verification). The root meta-class is the test model inheriting from and applying to the model meta-class. The test architecture concepts are related to the organization and realization of a set of related test cases. These include test context meta-class, which consist of one or more related test cases. The verdict of a test case is assigned by an arbiter. There is a default arbitration algorithm ordering the verdicts according to the hierarchy of their types. Similarly, test behaviour concepts describe the behaviour of the test cases that are defined within a test context. Associated with test cases are test objectives, which describe the capabilities the test case is supposed to validate. Test cases consist of behaviour, which includes validation actions. Validation actions update the verdict of a test case. They are created as user-defined blocks with Matlab functions behind them yet. Log actions, which write information about the tests are represented by an attribute (logSignalData) of a connection line called signal. Behavioural concepts also include the verdicts that are used to define the test case outcomes.

Simulink Test meta-model depends on the SL meta-model, while two additional meta-models, described in the next subsection, depend on the SLTest meta-model.

2.3 Additional Meta-models Overview

The additional Test Directive (TestDirective) and Environment Test Directive (EnvTestDirective) meta-models govern information gained from the test requirements and safety-critical requirements, respectively. TestDirective models are used only to refine the transformations from system model to test model according to the requirements or adding some implicit statements enhancing the mapping rules. Whereas EnvTestDirective models provide data on situations, when critical variables exceed/don't reach their defined values, some dangerous data ranges are achieved, unexpected situations or additional interference actions from the environment occur. These additional data must be treated as test data and enrich/refine the tests appropriately.
The proposed EnvTestDirective meta-model is developed using safety-critical requirements analysis. It gives the semantics overview and is defined in different packages with concept structures for EnvTestDirModel, Environment Noise, Critical Data, etc. It re-uses, similarly to the other test meta-models given in this paper, concepts provided within the SL meta-model (i.e. Model meta-class), but also SLTest meta-model (i.e. package containing Simulation Data).

3 Transformation on the Base of an Example

In this section, the example of transformations from a simple system model to test models representing executable test cases is depicted. Additionally, a model providing interferences from the environment is introduced.
The test objectives are connected with the definition of the functionality belonging to the validation actions. These are: checking of the quality of signals (continuous and discrete), looking for timeouts and matching of the test requirements with test model elements.
Our example model is restricted to two levels hierarchy. More complicated designs (including closed loops models) might cause additional problems or demand constraints for the transformation rules. It is also assumed that an *environment test directive model* is previously delivered as a Simulink design.
Some of the general transformation rules on the meta-models level used for the example are given below and explained after that:

```
pre: SystemModel::Element(n)→ TestModel::Element(n)
1.SystemModel::Subsystem→ TestModel::TestComponent
  || SystemModel::Subsystem→ TestModel::SUT
2.SystemModel::Number(Subsystem)→ TestModel::Number(TestModel)
3.SystemModel::Number(Subsystem)→ TestModel::Number(TestContext)
       if (!SUT) {
4.SystemModel::Block.Type(Inport)→TestModel::Block.Type(UserDefinedFun
  ctions.StartPathTestCaseTimer)
  && TestModel::PathTestCaseTimer.Value==[t(expectedExecution)+t(delay)]
5.SystemModel::Block.Type(Outport)→ TestModel::Block.Type(UserDefinedFun
  ctions.StopPathTestCaseTimer)
6.SystemModel::Block.Type(Outport)→ TestModel::Block.Type(UserDefinedFun
  ctions.ValidationAction)
7.SystemModel::Block.Type(Outport)→ TestModel::Block.Type(Sinks.Scope)
8.SystemModel::EnvTDModel→TestModel::Block.Type(UserDefinedFunctions.
  ValidationAction)}
```

3.1 Transformations from System Model to Test Models

Figure 2 presents a simple Simulink system *model*[1], which is to be converted into a set of *test models*. Continuous signal no. 1 coming from the sensor is flowing through a *line* from the *block* of type *Inport* to the *block* typed by *Demux*. Then the signal is split going to the *block* typed by *Integrator* on the one side and to the *block* typed by *Gain* on the other side. After that the Integrator's activity starts and the resulting signal becomes to be continuous signal no. 2 and is lead using the *line* connection to the *block* typed by a *subsystem*. The functions called inside the subsystem give continuous signal no. 3, which is introduced into the *block* typed by *Outport* – a link to the potential actuator. A similar procedure applies to the other paths in the model.

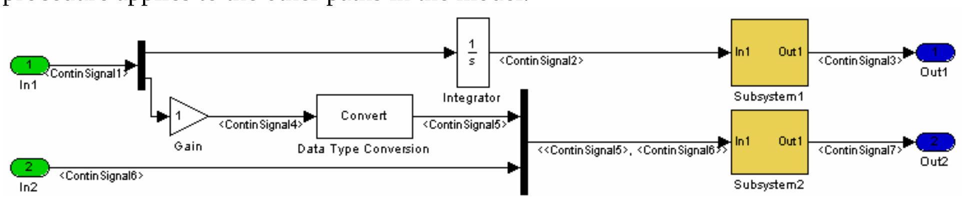


Figure 2: Simulink System *Model*

Let us consider a testing scenario of the given *model*. Firstly the structure of the system *model* is adopted to the *test model* (rule *pre*). Afterwards, test architecture artefacts are added. In Figure 3 Simulink *test model* derived from the system is shown. It deals with the situation when Subsystem1 is the *SUT* (rule 1). In another scenario the Subsystem2 could be treated as *SUT*, letting all the other *subsystems* on this level be a *test component*. Each *subsystem* which is not empty is the potential candidate to be the *SUT*. That is why the *test contexts* define all the possible combinations of *SUT* and *test components* with the rotating *SUT* role assignment. *Test contexts* have no explicit notation and each separate *test context* is expressed as a separate *test model* (rule 3). Hence, there are two instances of such *test models* for the considered example (rule 2). The next transformation step is to add the other test artefacts expressing behaviour to the *test model*. Two types of *test cases* are distinguished: *block test case* and *path test case*. Additionally, *path test case timer actions*, *clocks* – giving the simulation time and *validation actions* are attached.
Time measurement is used to assure proper termination of *test cases* [DS04] as well as appropriate duration of *block* execution. In order to assure a proper termination of a *path test case*, a *path test case timer* is started at the beginning of a *path test case*. Its duration is chosen to be slightly longer ($t_{PTcT} = t_{expected\ execution} + t_{delay}$) then the expected execution time of the *test case* (rule 4, 5). At the end of each possible test sequence, the considered *timer* is stopped. A *verdict* value expressing the incorrect behaviour is generated when the *path test case timer* expires. For each path describing model behaviour in the diagram of Figure 3 a timer is started at the beginning and stopped at the end of it. Simulink doesn't support such *path timer actions*, thus additional user-defined blocks delivering those functionalities must be still created. Further on, *validation action* is applied. *Validation actions* should update the *verdict* of a *test case*.

[1] Words written using cursive represent the *meta-classes* from the given meta-models, respectively.

They are created as user-defined *blocks*. There are two kinds of *validation actions – path validation action and block validation action. Path validation action* applies to the *path test case*. It checks for timing out of the timer and updates the *verdict* according to the other *verdicts* established on the path `(rule 6, 7, 8).`

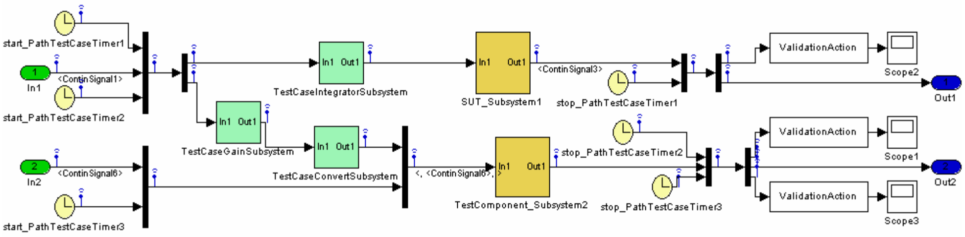


Figure 3: Derived *Simulink Test Model* for Subsystem1 being the *SUT*

Following our transformation, all the *blocks* present at the *model* or *subsystem* level in the considered system model example are handled. For such a particular block, a new subsystem in the *test model* is created so as to establish a so called *block test case*.
In every reactive system, the response time of an event should be restricted by a timer [DS04]. For this purpose, a *clock* is assigned to each *block* to measure its execution time. If the response is received within its expected duration, the *verdict* is set to the value expressing correct behaviour. Otherwise an alternative behaviour, i.e. *default* behaviour, is activated. Figure 4 presents insights of a *block test case* for Integrator subsystem. Herein a *clock* measuring the simulation time is attached to the *block* of type *Integrator*.

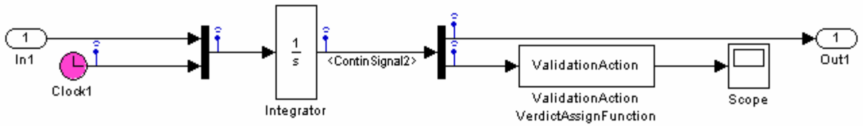


Figure 4: Derived *Block Test Case* for Integrator Encapsulated in a *Subsystem*

Block validation action is provided at the end of each *block test case*. Additionally to the *block test case*, a *reference model for block test case* is build. It includes almost all the elements as the former *test case* and is encapsulated in a *subsystem*, however in a separate *model* (see Figure 5).

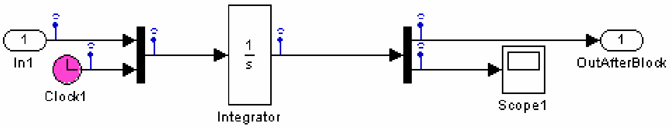


Figure 5: Derived *Subsystem* Put in *Reference Model for Block Test Case*

Block validation action should compare dynamically those two *subsystems'* outputs *(block test case subsystem* and *subsystem*, which is put in the *reference model for block test case)* after their simulation. *Reference model for block test case* is executed without any interferences, while *block test case* is executed having applied *EnvTestDirective model*. The results of the considered *block validation action* can be additionally observed by the application of the *Scope* block.

Reference model for block test case (Figure 5) for the considered *block test case* (Figure 4) has similar structure as the former one. However during simulation no interference data from the environment, no delays and no additional data from the requirements are provided. This enables to get both the pure and the real behaviour of the considered test case, so as to compare them and conclude about the verdict.

Simulation input data are needed for the functions continuum dynamic comparison. Data are derived from equivalence classes of their types and additional values given in the specification. This idea bases on CTM/ES method [CFS04] discussed in the next sections and is not concerned for the example yet.

The *block* typed by *validation action* must be intelligent enough to assure that some slight discrepancies between the pure and the real results are permissible on the one side, however safety critical requirements mustn't be ignored on the other side. Requirements are delivered by application of *EnvTestDirective model*, which enables to distinguish between soft and hard real-time requirements.

Additionally to the validation actions, *verdict* types and their hierarchy should be discussed. One of the proposals is to use fit/misfit for the arbitration of test purposes coming from the system requirements, pass/fail for the functional aspects, in-time/out-of-time for checking the duration and continuous-in-time/discrete-in-time for the time continuous aspects. Moreover, the interactions between the *verdict* types must be also considered in the future.

3.2 Additional Environmental Test Directive Models Derived from the Specification

Let us imagine an automatic control unit being activated in a given system. The safety-critical requirement specifies the following: *Always when any action from the environment (implied by a user or caused by other external action (i.e. external behaviour or changed value of some variable)) occurs, the control unit should be switched off/deactivated*.

The situation is modelled in Figure 6. On the meta-model level all the elements are put in the *EnvTestDirective model* meta-class, which inherits directly from SL *model* meta-class. The external action is represented by a *block* of type *Inport* (for SL meta-model), but also by *critical action* (for EnvTestDirective meta-model). *Inport* is connected via a *line* carrying the signal value with another *block* of type *Relational Operator*. *Block* of type *Constant* delivers the 0.0 value additionally to the action so as to let the operator compare the signal value with 0.0. If the external action value is bigger then 0.0, a *block* of type *trigger* is activated shutting off the control unit (a *subsystem*). Comparison of the two values and the result are encapsulated by *if* meta-class. The resulting action becomes to be the *body*. Finally, the deactivation of the control unit is a *critical action*.

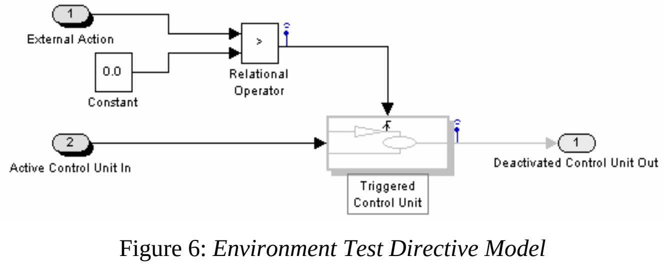


Figure 6: *Environment Test Directive Model*

This example shows that application of an *EnvTestDirective model* on an existing SL *model, system* or *subsystem* implies the *validation action* to be sensitive on the changes coming from the system outside.
As already mentioned, the transformation rules for the example in this paper are applied only manually. Due to missing technology, the rules implementation is not applicable for the large scale projects yet. However, further investigations on the subject promise that the presented solution may raise a lot of attention in the middle future.

4 Related Work

Several efforts have been undertaken to establish an approach to model-based test generation for embedded systems. Research and industrial work is being continuously developed. Algorithms have been defined to derive tests from formal system specification given in various notations.
One of the approaches is Time Partitioning Testing (TPT) [Leh00]. It is an approach for testing the dynamic functional behaviour of embedded systems. It supports the selection and documentation of test data on the semantic basis of so-called testlets and the syntactic techniques Direct Definition, Time Partitioning and Data Partitioning which are used to build testlets. Testlets facilitate an exact description of test data and guarantee the automation of test execution and test evaluation. The concepts of TPT correspond mainly to Environment Test Directive models specified using the system and test requirements in the approach presented in this paper.
Data Partitioning is strictly related to the next tool, called MTest [CFS04] and its Classification Tree Method for Embedded Systems (CTM/ES). This method contributes to testing of embedded systems very much. Using an interface description, which can be based on the specification and/or an executable model of the software, test scenarios can be derived systematically and described in a graphical way so as to provide the user with visual information about test coverage [Con04]. In our work, the data derivation from the equivalence classes of their types is provided with Simulink Test models, whereas catching of the interferences from the environment is given by EnvTestDirective models. Additionally, intermediate data flowing along the path are also treated as interfaces between different components (*subsystems*).
A similar approach to CTM/ES test data retrieval is realised by Reactis tool [Reactis]. The tests may be run on the models themselves to study and revise model behaviour. They may be applied to source-code implementations of models to ensure conformance with model behaviour. Also back-to-back tests are supported.
Matlab Automated Testing Tool (MATT) [Hen00] enables to create custom test data for model simulations or executables. Here basic functional tests on model level may be created and executed. Those issues are covered by EnvTestDirective models given in this paper.
Such tools, like EmbeddedValidator [BBS04] are considered for the discussed approach, as they validate and verify the model. The model checking technology is automatic and complete in a mathematical sense, meaning that it can detect every logical design flaw and error in the model being validated. However, it is assumed that model checkers are used before test generation.

MLIB/MTRACE and ControlDesk [dSpac99] tools make it possible to monitor, display and change the variables while the simulation is running. This is a good technological hint, how to apply the EnvTest Directive models onto SL test models.

Moreover, Simulink Test meta-model designed for embedded systems has been inspired by two factors: existence of UML 2.0 Testing Profile [U2TP04] for object-oriented software development as well as by MDA-driven testing [ZDS$^+$05]. They are both related to this research.

5 Outlook and Future Work

The aim of this paper is to demonstrate that it is worth to apply MDA artefacts for Simulink models and their testing. For this purpose Simulink, Simulink Test, Environment Test Directive and Test Directive meta-models are introduced and presented in brief. An example of retrieving the executable tests from Simulink models is provided. Some simple transformation rules are depicted. Corresponding meta-classes are assigned to the elements of the models. Further on, work related to the subject regarding automotive domain is given. Finally, conclusions are taken and further work challenges are outlined.

All the meta-models need a lot of improvement. Their structure is still in the development phase in the ongoing research projects. The tools adapters must be built to be able to interpret the Simulink models and store them in the repository at the one side, as well as to retrieve the generated Simulink Test models from the repository and expose them correctly in Simulink at the other side. Derivation of executable test models is possible by applying some modelling restrictions. They should be collected in the future. Further transformation rules have to be formalised and implemented. Some tools [IKV], [EMF] already enable transformations between models on the meta-model level. One of the environments being often used to demonstrate the feasibility of transformations is Eclipse [Eclipse] with its Eclipse Modelling Framework (EMF) plug-in. The meta-models can be defined by Rational Rose tool in UML. The EMF generator can create a corresponding set of Java implementation classes from such a Rose model. The mapping rules can be realised via EMF Java API. The transformations can generate objects within a Simulink Test meta-model, which could enable the execution of the tests directly in Simulink. The transformation rules could be also formalized using Query/View/Transformation (QVT) [QVT04] language. However the tools, like Borland Together [BT06] or Tefkat [SL04], both enabling QVT, are still very limited and incomplete to perform such formalism.

Continuing research will focus also on formal models of computation [LN04] found behind the presented models to let define formally-proven transformation rules for some complicated cases. Moreover, tool-independent ideas for meta-modelling of hybrid real-time systems will be considered. This would give a generic view on the development and testing in this area, taking into account real-time, continuous signals. Any other tools like SCADE [BDK05] used for embedded software development could be enriched with such generic test automation ideas. Additionally, requirements-driven automatic test information retrieval in the context of EnvTestDirective models must be investigated (i.e. by help of temporal logic). Also a facility to manage the obtained test results should be provided in the finishing phase of the work.

Last, but not least the autonomic communication paradigms could contribute to the self-organisation of the system model enabling the automatic generation of the executable test models. Simulink model elements could interpret themselves and re-build them so as to achieve the expected goals. Furthermore, test-driven improvements of the system under test may be established. For example, when the variable value is reaching a critical range, the system would organise itself to avoid the dangerous value by usage of some intelligent rules, that have to be defined.

References

[BBS04] Bienmüller T., Brockmeyer U., Sandmann G., Automatic Validation of Simulink/ Stateflow Models, Formal Verification of Safety-Critical Requirements, 2004, Stuttgart

[BDK05] Bouali A., Dion B., Konishi K., Using Formal Verification in Real-time Embedded Software Development, JSAE 2005

[BT06] Borland Together 2006 - http://www.borland.com/downloads/download_together.html

[CFS04] Conrad M., Fey I., Sadeghipour S., Systematic Model-Based Testing of Embedded Control Software: The MB3T Approach, Edinburgh, May 25th, 2004, ICSE 2004

[Con04] Conrad M., A Systematic Approach to Testing Automotive Control Software, Convergence Transportation Electronics Association 2004

[dSpac99]MLIB, http://www.dspace.de/ftp/patches/NFMG/SfC32/NewFeaturesAndMigration.pdf

[DS04] Dai Z.R., Schieferdecker I.: Time Concepts for UML 2.0 Based Testing. SIVOES 2004, RTAS 2004, Toronto, Canada, May 2004

[Eclipse] Eclipse Platform: http://www.eclipse.org/platform/

[EMF] Eclipse Modelling Framework: http://www.eclipse.org/emf/

[Hen00] Henry J., Automation: The Key to Improving Testing in the Maintenance of Real-time Systems, Submission of a Position Paper for WESS 2000, University of Montana

[IKV] IKV++ Technologies, medini MM tool, http://www.ikv.de/

[Lee04] Lee E. A., Actor-Oriented Design: A focus on domain-specific languages for embedded systems, MEMOCODE'2004, San Diego, California, 2004

[Leh00] Lehmann E., Time Partition Testing: A Method for Testing Dynamic Functional Behaviour, TEST2000, London, UK, 2000

[LN04] Lee E. A., Neuendorffer S; Concurrent Models of Computation for Embedded Software, TM UCB/ERL M04/26, University of California, Berkeley, CA 94720, July 22, 2004

[MatML] Mathworks, Matlab/Simulink/Stateflow, http://www.mathworks.com/products/matlab/

[MDA03]OMG: Model-Driven Architecture (MDA), http://www.omg.org/docs/omg/03-06-01.pdf

[MOF04]OMG: MOF, v1.4, http://www.omg.org/technology/documents/formal/mof.htm

[NK04] Neema S., Karsai G., Embedded Control Systems Language for Distributed Processing (ECSL-DP), Vanderbilt University, ISIS-04-505, 2003-2004

[QVT04] OMG: MOF Query/Views/Transformations, 2nd Revised Submission, ad/04-01-06

[Reactis] Reactis Tester Tool, http://www.reactive-systems.com/tester.msp

[SD04] Schieferdecker I., Din G.: A meta-model for TTCN-3. 1st International Workshop on Integration of Testing Methodologies, ITM 2004, Toledo, Spain, Oct. 2004.

[SL04] Steel J., Lawley M., Model-Based Test Driven Development of the Tefkat Model-Transformation Engine, ISSRE 2004, pp. 151-160

[U2TP04]OMG: UML 2.0 Testing Profile. Final Adopted Specification, ptc/04-04-02, 2004

[VML97]IABG: Das V-Modell-Entwicklungsstandard für IT-Systeme des Bundes, Vorgehensmodell, Kurzbeschreibung, 1997, http://www.v-modell.iabg.de/

[ZDS+05]Zander J., Dai Z. R., Schieferdecker I., Din G., From U2TP Models to Executable Tests with TTCN-3 - An Approach to Model Driven Testing, Canada, Montreal, TestCom’05

Technische Universität Braunschweig
Informatik-Berichte ab Nr. 2002-04

2002-04	H. G. Matthies, J. Steindorf	Partitioned Strong Coupling Algorithms for Fluid-Structure-Interaction
2002-05	H. G. Matthies, J. Steindorf	Partitioned but Strongly Coupled Iteration Schemes for Nonlinear Fluid-Structure Interaction
2002-06	H. G. Matthies, J. Steindorf	Strong Coupling Methods
2002-07	H. Firley, U. Goltz	Property Preserving Abstraction for Software Verification
2003-01	M. Meyer, H. G. Matthies	Efficient Model Reduction in Non-linear Dynamics Using the Karhunen-Loève Expansion and Dual-Weighted-Residual Methods
2003-02	C. Täubner	Modellierung des Ethylen-Pathways mit UML-Statecharts
2003-03	T.-P. Fries, H. G. Matthies	Classification and Overview of Meshfree Methods
2003-04	A. Keese, H. G. Matthies	Fragen der numerischen Integration bei stochastischen finiten Elementen für nichtlineare Probleme
2003-05	A. Keese, H. G. Matthies	Numerical Methods and Smolyak Quadrature for Nonlinear Stochastic Partial Differential Equations
2003-06	A. Keese	A Review of Recent Developments in the Numerical Solution of Stochastic Partial Differential Equations (Stochastic Finite Elements)
2003-07	M. Meyer, H. G. Matthies	State-Space Representation of Instationary Two-Dimensional Airfoil Aerodynamics
2003-08	H. G. Matthies, A. Keese	Galerkin Methods for Linear and Nonlinear Elliptic Stochastic Partial Differential Equations
2003-09	A. Keese, H. G. Matthies	Parallel Computation of Stochastic Groundwater Flow
2003-10	M. Mutz, M. Huhn	Automated Statechart Analysis for User-defined Design Rules
2004-01	T.-P. Fries, H. G. Matthies	A Review of Petrov-Galerkin Stabilization Approaches and an Extension to Meshfree Methods
2004-02	B. Mathiak, S. Eckstein	Automatische Lernverfahren zur Analyse von biomedizinischer Literatur
2005-01	T. Klein, B. Rumpe, B. Schätz (Herausgeber)	Tagungsband des Dagstuhl-Workshop MBEES 2005: Modellbasierte Entwicklung eingebetteter Systeme
2005-02	T.-P. Fries, H. G. Matthies	A Stabilized and Coupled Meshfree/Meshbased Method for the Incompressible Navier-Stokes Equations — Part I: Stabilization
2005-03	T.-P. Fries, H. G. Matthies	A Stabilized and Coupled Meshfree/Meshbased Method for the Incompressible Navier-Stokes Equations — Part II: Coupling
2005-04	H. Krahn, B. Rumpe	Evolution von Software-Architekturen
2005-05	O. Kayser-Herold, H. G. Matthies	Least-Squares FEM, Literature Review
2005-06	T. Mücke, U. Goltz	Single Run Coverage Criteria subsume EX-Weak Mutation Coverage
2005-07	T. Mücke, M. Huhn	Minimizing Test Execution Time During Test Generation
2005-08	B. Florentz, M. Huhn	A Metamodel for Architecture Evaluation
2006-01	H. Giese, B. Rumpe, B. Schätz (Herausgeber)	Tagungsband des Dagstuhl-Workshop MBEES 2006: Modellbasierte Entwicklung eingebetteter Systeme II