MBEES
MBEES
MBEES
MBEES
MBEES

Tagungsband des Dagstuhl-Workshops

Modellbasierte Entwicklung eingebetteter Systeme

Torsten Klein
Bernhard Rumpe
Bernhard Schätz

carmeq
software & systems

SCHLOSS DAGSTUHL
INTERNATIONALES
BEGEGNUNGS-
UND FORSCHUNGSZENTRUM
FÜR INFORMATIK

Software
Systems
Engineering

Tagungsband

Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme

WS-Nr. 05022
Model-Based Development of Embedded Systems)

10. – 14.01.2005

Informatik Bericht
TU Braunschweig
2005-01

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

Inhaltsverzeichnis

Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme

(WS-Nr. 05022 Model-Based Development of Embedded Systems)

Die modellbasierte Entwicklung eingebetteter, softwarebasierter Systeme beruht auf der anwendungsorientierten Modellierung der zu realisierenden Systeme sowohl unter Nutzung von Standardsprachen, wie der UML und deren Elementen, wie Komponenten, Nachrichten, oder Zustände, als auch von domänenspezifischen Konzepten, wie etwa zeitbehafteten Ereignissen oder Signalen, synchronem oder asynchronem Datenfluss und Priorisierungs- und Unterbrechungskonzepten. Durch den Einsatz anwendungsorientierter statt codezentrierter Modelle können Aspekte der Implementierung (z.B. Bus- oder Task-Schedules, Implementierungstypen) abstrahiert werden, während besonders wichtige und kritische Aspekte explizit und frühzeitig modelliert werden (z.B. Zeit, Prioritäten oder Kommunikationsaspekte). Die Anwendung analytischer und generativer Verfahren auf diesen Modellen erlaubt die effiziente Entwicklung hochqualitativer Software.

Modellbasierte Vorgehensweisen mit der verstärkten Trennung von anwendungs- und implementierungsspezifischen Modellen haben in der Softwareentwicklung in den letzten Jahren zunehmend an Bedeutung gewonnen. Dies zeigt sich einerseits in domänenspezifischen Ansätzen, die mit aufgabenspezifischen Modellen (z.B. synchroner Datenfluss) und dazugehörigen Werkzeugen spezialisierte Bereiche (z.B. Regelungs- und Steuerungsalgorithmen, Anlagensteuerung) bedienen. Auf der anderen Seite werden mit allgemeinen Modellen (z.B. Zustand-Ereignis-Modellen) und dazugehörigen Werkzeugen, oder auch mit der noch wenig erprobten Model Driven Architecture (MDA) auf der Basis von meist objektorientierter Architekturmodelle zunehmend Anstrengungen unternommen, in den Bereich domänenspezifischer Modellierung vorzustoßen. Hier zeigt sich, dass insbesondere durch Verbesserung der Entwicklungswerkzeuge, vor allem hinsichtlich Implementierungsqualität, Bedienkomfort und Analysemächtigkeit, das Paradigma der modellorientierten Softwareentwicklung auch im Bereich eingebetteter Systeme an Einfluss gewinnen.

Dabei wird die Bedeutung anwendungsorientierter Modelle mit domänenspezifischen Konzepten für diese Ansätze oft unterschätzt. Daher steht insbesondere dieser Aspekt in diesem Workshop im Vordergrund. Die in diesem Workshop-Tagungsband zusammengefassten Papiere stellen zum Teil gesicherte Ergebnisse, Work-In-Progress, industrielle Erfahrungen und innovative Ideen aus diesem Bereich zusammen. Damit sind wesentliche Ziele dieses Workshops erreicht:

- Austausch über domänenspezifische Probleme und existierende Ansätze zwischen den unterschiedlichen Disziplinen (insbesondere Elektro- und Informationstechnik, Maschinenwesen/Mechatronik, und Informatik)
- Austausch über relevante Probleme in der industriellen Anwendung 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 die im Wesentlichen sehr gut abgedeckt sind, fokussieren auf Teilaspekte der modellbasierten Entwicklung eingebetteter Softwaresysteme. Darin enthalten sind unter anderem:

- Domänenspezifische Ansätze zur Modellierung von Systemen (z.B. aus der Luft- und Raumfahrt, Bahntechnik, Automobilindustrie sowie der Produktions- und Automatisierungstechnik)
- Integration ereignisgesteuerter und zeitgesteuerter 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 der Industrie, von Werkzeugherstellern und aus der Wissenschaft der Auf- und Ausbau einer für alle Beteiligten gewinnbringenden Community gestartet wird, auf der sich eine solide Basis zur Weiterentwicklung des noch jungen Themenbereiches der modellbasierten Entwicklung eingebetteter Systeme etablieren lässt.

Die Durchführung eines für alle Beteiligten erfolgreichen Workshops ist ohne vielfache Unterstützung nicht möglich. Wir danken daher den Mitgliedern des Programmkomitees, den Mitarbeitern von Schloss Dagstuhl, unseren Sponsoren, aber insbesondere Herrn Krahn von der TU Braunschweig, für die helfende Unterstützung bei der Durchführung des Reviewprozesses, der Organisation der Webseite und der Zusammenstellung der Proceedings.

Schloss Dagstuhl im Januar 2005,

Das Organisationskomitee

Torsten Klein, Carmeq GmbH Berlin

Bernhard Rumpe, TU Braunschweig

Bernhard Schätz, TU München

Programmkomittee

Ulrich Freund, ETAS GmbH,
Michaela Huhn, TU Braunschweig
Andrew Lyons, IBM Rational
Klaus D. Müller-Glaser, Uni Karlsruhe
Alexander Pretschner, ETH Zürich
Matthias Riebisch, TU Illmenau
Wilhelm Rossak, Uni Jena
Andreas Schürr, Uni Darmstadt
Birgit Vogel-Heuser, Uni Wuppertal

Carmeq konzipiert, entwickelt und integriert softwarebestimmte Systeme für die Automobilindustrie. Zusammen mit Herstellern und Zulieferern bewältigt Carmeq die Komplexität innovativer Systeme und Architekturen, optimiert mit Hilfe fortschrittlicher Technologien und Prozesse Unternehmensabläufe und entwickelt spezifische Software für Automobilelektronik. Als Tochter des Volkswagen-Konzerns arbeitet Carmeq für die gesamte Automobil- und Zulieferindustrie.

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.

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.

Von Use Cases zu Test Cases: Eine systematische Vorgehensweise

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

Kurzfassung

Anwendungsfälle (Use Cases) dienen oftmals nicht nur als Grundlage für den Systementwurf, sondern auch für System- und Abnahmetests. Die Ableitung der Testfälle geschieht jedoch oft intuitiv und unsystematisch. In dieser Arbeit beschreiben wir eine Methode zur systematischen Erzeugung von Testfällen aus Anwendungsfällen. In einem ersten Schritt wird der semantische Bezug zwischen Use Case Elementen und Systemfunktionen hergestellt. In einem zweiten Schritt werden die Use Cases aufbereitet und in Aktivitäts-Diagramme überführt, die dann mit automatischen Testgenerierungswerkzeugen weiter verarbeitet werden können. Im Gegensatz zu Ansätzen, die auf der vollautomatischen Analyse natürlicher Sprache basieren, erlaubt unsere Vorgehensweise, alle normalerweise benötigten Sprachelemente zu verwenden. Im Gegensatz zu informellen oder leitfadenbasierten Methoden kann unsere Methode gut durch automatisierte Werkzeuge unterstützt werden.

1 Einleitung

Ein wesentliches Merkmal der modellbasierten Entwicklung eingebetteter Steuergeräte ist es, dass die Qualitätssicherung parallel zum gesamten Entwurfsprozess durchgeführt wird. Bereits frühzeitig im modellbasierten Entwicklungsprozess wird aus den informell formulierten Anforderungen ein ausführbares Modell erstellt, welches dann in mehreren Schritten bis zu einem Implementierungsmodell verfeinert wird. Parallel dazu erfolgt die Erstellung ausführbarer Testfälle, mit denen das Modell in jedem Reifestadium getestet wird.

Für die Ableitung der Testfälle gibt es dabei mehrere Möglichkeiten. Zum einen kann das Modell selbst genutzt werden, um daraus Testsequenzen zu generieren (siehe z.B. [SHS03]). Aus einer Simulation des Systemmodells zusammen mit einem Modell der vor-

gesehenen Systemumgebung werden Eingangsdaten für die Sensoren generiert und die zu erwartenden Reaktionen an den Aktuatorausgängen gemessen. Die so gewonnenen Ereignisfolgen werden für den Test der nächsten Entwicklungsstufe verwendet. Sie enthalten die Stimuli für das zu testende System und dienen gleichzeitig als Testorakel für das Systemverhalten. Diese Vorgehensweise bietet sich insbesondere im letzten Entwicklungsschritt an: Aus dem Implementierungsmodell werden so Testfälle für Hardware-in-the-Loop Tests gewonnen, mit denen das korrekte Zusammenspiel der generierten Software mit dem eingebetteten Zielprozessor untersucht wird.

Eine andere Möglichkeit, Testfälle zu erhalten, besteht darin, die ursprünglichen Anforderungsbeschreibungen zu verwenden. Aus den funktionalen Benutzeranforderungen werden dabei (möglichst systematisch) unmittelbar Testfälle erzeugt, mit denen die ausführbaren Modelle getestet werden. Auf diese Weise kann bereits das allererste grobe Architekturmodell systematisch auf Übereinstimmung mit bestimmten Anforderungen überprüft werden. Ebenso wie das Systemmodell selbst unterliegen bei dieser Methode auch die Testfälle einer Entwicklung und Anpassung an die einzelnen Entwicklungsstufen. Die Vorgehensweise unterstützt vor allem System- und Abnahmetests, da die Benutzersicht auf das Gesamtsystem im Vordergrund steht.

Ein wichtiger Punkt bei dieser Vorgehensweise ist die Systematik der Erstellung von Tests aus den im Pflichtenheft beschriebenen Anforderungen. Zur Beschreibung funktionaler Anforderungen werden beim objektorientierten Softwareentwurf oftmals *Use Cases* verwendet. Für geschäftsprozessunterstützende Softwaresysteme hat sich insbesondere die in [Coc00] und [SW01] definierte Darstellungsform durchgesetzt. Ein Use Case ist dabei eine Beschreibung typischer Nutzer-System-Interaktionen in natürlicher Sprache oder tabellarischer Notation. Use Cases sind oftmals Teil des Kontraktes zwischen Auftraggeber und Auftragnehmer und bilden daher eine Grundlage für die Systementwicklung. Im Bereich eingebetteter Systeme werden Pflichtenhefte dagegen vielfach nicht direkt in Form von Use Cases formuliert, sondern durch Mischformen aus tabellarischen und informellem Text. Implizit sind jedoch auch solche Dokumente häufig durch die Anwendersicht strukturiert und können daher zur Erstellung von Use Cases genutzt werden. In [DPB03] sind Richtlinien zur Erstellung von Use Cases aus informellen Anforderungsbeschreibungen für eingebettete Systeme angegeben.

Use-Case-Beschreibungen lassen auch als Ausgangspunkt für funktionale Systemtests verwenden. Da die Formulierung von Use Cases jedoch in natürlicher Sprache erfolgt, ist es beim heutigen Stand der Technik nicht möglich, sie vollständig automatisch in Testfälle zu transformieren. Es existiert zur Zeit nicht einmal ein standardisiertes Format, in dem Use-Case-Beschreibungen notiert werden. Ein wichtiges Problem ist daher die Aufbereitung von Use-Case-Beschreibungen für den Systemtest.

Zur Lösung dieses Problems existieren mehrere Ansätze. Zum einen gibt es Versuche, die Ausdrucksmächtigkeit natürlicher Sprachen einzuschränken [Sch98]. Zum anderen können Use Cases auf relevante Formulierungen und Schlüsselwörter untersucht werden. Bei der (manuellen) Erstellung von Testfällen kann diese Information benutzt werden [McC03].

In dieser Arbeit schlagen wir eine interaktive Vorgehensweise zur Aufbereitung von Use-

Case-Beschreibungen vor, die diese beiden Ansätze vereint. Zunächst werden den einzelnen Schritten im Use Case die entsprechenden Systemfunktionen und -reaktionen zugeordnet und der Kontrollfluss formalisiert. Anschließend werden die Use Cases in eine formalere Notation überführt, von der aus sie mit automatischen Testfallgenerierungsalgorithmen weiterverarbeitet werden können.

2 Use-Case-Beschreibungen und ihre Formalisierung

Bei der Anforderungsanalyse großer Softwaresysteme werden in der Regel Teams von Experten aus den unterschiedlichsten Fachrichtungen eingesetzt, die parallel und verteilt einzelne Use Cases erstellen und zur Menge der Anforderungen hinzufügen. Daraus ergeben sich oft *inkonsistente*, *mehrdeutige* und *unvollständige* Spezifikationen, die nicht zur automatischen Testfallerzeugung genutzt werden können [Pos96].

Daher müssen textuelle Anforderungen in Use Cases in semantisch eindeutiger Weise formalisiert werden, bevor daraus automatisch Testfälle erstellt werden können. An eine Methodik zum Formalisieren von Anforderungen werden unterschiedliche Ansprüche gestellt, abhängig davon, ob das Ziel die Entwicklung oder der Test des Systems ist. Werden Anforderungen für die Systementwicklung formalisiert, dürfen dabei noch keine Entwurfsentscheidungen getroffen werden. Zwischen informellen und formalisierten Anforderungen darf keine Verfeinerungsbeziehung erzeugt werden, wie sie zwischen Anforderungen und Design besteht. Die formalisierten Anforderungen müssen noch jede mögliche Realisierung zulassen, welche die informell notierten Anforderungen des Auftragebers an das System erfüllt.

Das Ziel des Systemtests ist es zu validieren, ob eine konkrete Realisierung die gestellten Anforderungen erfüllt. Mit der zu prüfenden Implementierung muss nur eine mögliche Verfeinerung der Spezifikation betrachtet werden – alle Entwurfsentscheidungen sind schon gefallen. Dementsprechend muss eine Methodik zur Formalisierung von Anforderungen für den Systemtest obige Forderung nicht erfüllen. Bei der Formalisierung für den Test lassen sich sogar design- und implementierungsspezifische Informationen aus dem Systementwurf für die Formalisierung nutzen.

Die Kernforderungen an die Formalisierung für den Systemtest lassen sich folgendermaßen zusammenfassen:

1. Beseitigung bzw. Verringerung des Interpretationsspielraumes, sowohl für den Kontrollfluss als auch für die einzelnen Schritte
2. Herstellung des Bezuges zur Implementierung
3. Erhaltung der expliziten Repräsentation der Szenarien

Als potentielles Zielformat für die Formalisierung von Use-Case-Beschreibungen sind prinzipiell alle Formalismen zur Darstellung von Interaktionen geeignet. Typische Vertreter dieser Kategorie sind *Message Sequence Charts* (MSC) [OMG03a], UML2.0 MSC

[OMG03b], *Life Sequence Charts* LSC [DH01], Activity-Diagramme [OMG03a] und Statecharts [Har87], [OMG03a].

Darüber hinaus gibt es Beschreibungstechniken, die speziell auf die Darstellung szenarienbasierter Nutzer-System-Interaktionen ausgerichtet sind, wie Use-Case-Schrittgraphen [Win99], Templates mit strukturiertem Text [Rup02], glossarbasierte Templates [RH04] und Strukturierungsvorgaben für Activity-Diagramme [HVFR04].

Nicht alle dieser Beschreibungstechniken eignen sich als Zielformat für die Formalisierung von Use-Case-Beschreibungen für den Systemtest, sofern obige Forderungen erfüllt werden sollen. Strukturierter Text ist nicht geeignet, da er einerseits viel zu aufwendig zu erzeugen ist, andererseits aber auch nur ein Zwischenformat ist. In Statecharts sind die in Use-Case-Beschreibungen repräsentierten Szenarien zwar noch implizit enthalten aber nicht mehr explizit dargestellt. In MSC lassen sich Szenarien zwar explit repräsentieren, jedoch nur mit linearen Kontrollflüssen.

In den UML2.0 MSC ist diese Einschränkung beseitigt worden. Eine weitere geeignete Repräsentationsform sind Activity-Diagramme, insbesondere bei Verwendung von Strukturierungsvorgaben, wie sie in [HVFR04] dargestellt sind.

Ein Großteil der in der Literatur dargestellte Überführungsverfahren von Use-Case-Beschreibungen fokussiert sich auf eine Formalisierung für die Systementwicklung. Wesentliches Ziel dieser Verfahren ist es, die textuelle Ausgangsspezifikation in eine formalere Darstellung zu überführen, welche fortan ausschließlich als Basis des Entwicklungsprozesses verwendet wird. In diese Kategorie fallen die meisten Verfahren, welche auf semantischer Textanalyse basieren und strukturierten Text erzeugen. Weiterhin existieren richtlinenbasierte Verfahren zur manuellen Überführung z.B. in Statecharts [DKvK+02]. Einige Formalisierungsmethodiken sind auch speziell auf den Systemtest ausgerichtet, z.B. das in [RG99] dargestellte Statechart-basierte Verfahren.

3 Systematische Überführung von Use Cases in Test Cases

In der modellbasierten Entwicklung [OMG04] wird zwischen plattformunabhängigen Modellen (PIM) und plattformspezifischen Modellen (PSM) unterschieden, aus welchen mithilfe mehrstufiger Transformationen der Code generiert wird, siehe Abbildung 1.

Im Test lassen sich ebenfalls plattformunabhängige Testfälle (PIT) und plattformspezifische Testfälle (PST) unterscheiden, in der Literatur auch oft als *logische* und *konkrete Testfälle* bezeichnet [SL02]. Die PST können anschließend in ausführbare Testskripte transformiert werden.

Use Cases beschreiben design- und technologieunabhängig typische Nutzerinteraktionen. Bei der Erstellung des PIM werden im Vergleich zu den Use Cases bereits erste Designentscheidungen getroffen. Zum Erstellen von PIT ist Wissen über das gewählte Design notwendig. Deshalb lassen sich Testfälle nicht allein aus den Use Case ableiten, sondern Wissen über die realisierenden Systemfunktionen und verwendeten Datentypen ist erforderlich. In dem hier vorgestellten Verfahren wird interaktiv der Bezug zwischen den se-

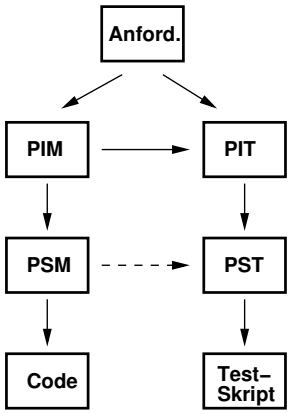

Abbildung 1: Formalisierung von Anforderungen als Transformation im Model-Driven-Testing

mantisch äquivalenten Schritten der Use-Case-Beschreibungen und den parametrisierten Systemfunktionen hergestellt. Anschließend wird diese Information genutzt, um die Use Cases in stereotypisierte Activity-Diagramme zu überführen.

Use Case als Ausgangspunkt. Das Verfahren wird am Beispiel des Use Cases *Record a Message* aus [PL99] dargestellt. Dort wird die UML-basierte Entwicklung eines digitalen Sound-Recorders beschrieben.

1. `The user selects a message slot from the message directory.`
2. `The user presses the 'record' button.`
3. `If the message slot already stores a message, it is deleted.`
4. `The system starts recording the sound from the microphone until the user presses the 'stop' button, or the memory is exhausted.`

Im Gegensatz zu Beispielen aus der Telekommunikation oder Geschäftsprozessen gibt es in eingebetteten Systemen nur ein eingeschränktes Interaktionsverhalten. Die Komplexität entsteht dabei vor allem durch die Parameter der Interaktion und nicht durch komplizierte Kontrollflüsse. Da allerdings eingebettete und kommunizierende Systeme zunehmend zusammenwachsen, sind die Grenzen fließend.

Bestimmung der Systemfunktionen und -reaktionen. Jegliche Nutzer-System-Interaktion, d.h. sowohl Systemfunktionen und -reaktionen, müssen über die Benutzerschnittstelle übertragen werden. Sofern die Benutzerschnittstelle nur durch ein *Graphical User Interface* (GUI) realisiert ist, genügt eine systematische Analyse der Oberfläche zur Bestimmung von Systemfunktionen und -reaktionen. In eingebetteten Systemen umfasst die

Systemschnittstelle zusätzlich Sensoren und Aktuatoren. Für diese gibt es häufig standardisierte Zugriffsfunktionen, die sich aus dem Systemmodell ablesen lassen.

Jedes ermittelte Element der Systemschnittstelle lässt sich mindestens einer Funktionalität zuordnen. Elemente, welche Eingaben des Systems aufnehmen (Buttons, Eingabefelder, Sensoren etc.) können den Systemfunktionen zugeordnet werden. Rein darstellende Elemente in graphischen Benutzeroberflächen (Fenster, Ausgabedialoge etc.) sowie Aktuatoren werden den Systemreaktionen zugeordnet.

Aus der Realisierung des Beispiel-Use-Case *Record a Message* lassen sich folgende Systemfunktionen ermitteln:

```
Systemfunktionen
================
select_MessageSlot(Slot)
start_Recording()
stop_Recording()
```

In einer konkreten Realisierung werden die abstrakten Systemfunktionen *start_Recording()* und *stop_Recording()* durch den Druck auf die entsprechenden Knöpfe aufgerufen. Es ist jedoch durchaus denkbar, dass auf diese Funktionen auch über andere Wege zugegriffen werden kann, z.B. über eine Fernbedienung oder ein Signal auf einem Multimedia-Bus.

Die aus der Realisierung des Beispiel-Use-Case bestimmten Systemfunktionen sind folgende:

```
Systemreaktionen
================
delete_MessageSlot(Slot)
record_Message(Slot)
```

Ermitteln des Kontrollflusses. Die Formalisierung der Use-Case-Beschreibungen lässt sich in zwei Aspekte trennen. Zum einen erfolgt die Formalisierung des Kontrollflusses, zum anderen die Verbindung der einzelnen Schritte mit den Systemfunktionen und -reaktionen.

Der Kontrollfluss wird zum Teil durch die Struktur des Templates für die Use-Case-Beschreibungen vorgegeben. Oft sind Teile des Kontrollflusses nur textuell beschrieben. Folgende Arten von Kontrollflüssen treten typischerweise in Use-Case-Beschreibungen auf: sequentielle Abfolgen, Schleifen, Fallunterscheidungen, alternative Abläufe, Sprünge sowie Includes weiterer Use Cases.

In einem Use Case Metamodell z.B. [RA98, Figure 3] werden die unterschiedlichen Kontrollflüsse als Realisierungen von *Flow of Actions* repräsentiert. Jedes dieser Konzepte repräsentiert Verbindungen zwischen Schritten, wobei die Anzahl der verbunden Schritte variiert. So werden beispielsweise in einer sequentiellen Abfolge nur jeweils zwei Schritte über eine Vorgänger-Nachfolger-Relation in Beziehung gesetzt. Eine bedingte Verzweigung hingegen verbindet drei Schritte: den Ausgangsschritt, welcher die Bedingung ent-

hält, und die beiden Schritte, mit welchen bei Erfüllung bzw. Nichterfüllung der Bedingung fortgesetzt wird.

Wie schon erwähnt, werden einige dieser Konzepte, z.B. sequentielle Abfolgen und alternative Abläufe, unmittelbar durch die templatebasierte Struktur repräsentiert und lassen sich entsprechend direkt aus der Struktur der Use-Case-Beschreibung ableiten. Andere, textuell repräsentierte Konzepte lassen sich jedoch nicht ohne weiteres ableiten, d.h. ohne Interpretation des Textes.

Ziel der Formalisierung des Kontrollflusses ist es, die einzelnen Schritte der Use-Case-Beschreibungen durch diese Konzepte zu verbinden, d.h. ein *konzeptionelles Modell* aufzubauen. Dieses kann entweder manuell erstellt werden oder werkzeuggestützt interaktiv aufgebaut werden [Mad04] [Fri04].

Schritt	Typ	Funktion oder Reaktion
1	F	select_MessageSlot(Slot)
2	F	start_Recording()
3	R	delete_MessageSlot(Slot)
4a	R	record_Message()
4b	F	stop_Recording()
4c	I	memory_exhaust

Abbildung 2: Beziehungen zwischen Schritten und Systemfunktionen und -reaktionen

Abbilden von Systemfunktionen auf Use-Case-Schritte. Jeder Use Case besteht aus Schritten, welche einen (bei sequentieller Abfolge) oder mehrerere weitere Schritte (bei Fallunterscheidungen, Schleifen, etc.) als Nachfolger haben können. Nachdem beim Ermitteln des Kontrollflusses der Zusammenhang der Schritte untereinander festgestellt wurde, wird nun der Inhalt der einzelnen Schritte betrachtet.

Im Black-Box-Systemtest wird die Eingabe-Ausgabe-Konformität zwischen Systemspezifikation und Systemrealisierung geprüft. In den Testfällen ist festzulegen, was für Eingaben in welcher Reihenfolge durch den Nutzer zu tätigen sind, einschließlich der zugehörigen Systemreaktionen.

Die Abfolge ist durch den Kontrollfluss bestimmt. Nun gilt es noch, die anderen Aspekte eindeutig zu bestimmen, d.h. festzustellen, welcher Aktor welche Systemfunktion in einem Schritt aufruft und welche Systemreaktionen dadurch hervorgerufen werden. Im Hinblick auf den Systemtest ist es ausreichend, einen Schritt entweder als eine Festlegung der auszuführenden parametrisierten Systemfunktion einschließlich dem ausführenden Aktor oder aber als parametrisierte Systemreaktion zu betrachten. Ein darüber hinausgehende Analyse ist aus Sicht des Systemtests ist nicht notwendig.

In dem als Beispiel dienenden Use Case *Record a Message* werden alle Systemfunktionen von dem gleichen Aktor *User* genutzt. In dem in Abbildung 2 dargestellten Ergebnis der Zuordnung ist deshalb der Aktor nicht mehr explizit aufgeführt. Die Spalte *Schritt* referenziert die Use-Case-Schritte. Die Spalte *Typ* bezeichnet den Typ des Schrittes, wobei

„F“ für Systemfunktion, „R“ für Systemreaktion und „I“ für einen internen Schritt steht (in diesem Fall für eine Ausnahme).

Erstellen der Zwischenrepräsentation und Testfälle. Im nächsten Schritt wird das *konzeptionelle Modell* in eine Zwischenrepräsentation übertragen. So wird beispielsweise das Konzept der sequentiellen Abfolge von Schritten in Use-Case-Beschreibungen in Activity-Diagrammen als zwei aufeinanderfolgende Activities dargestellt.

Das Ergebnis der Formalisierung des Beispiels ist in Abbildung 3 dargestellt.

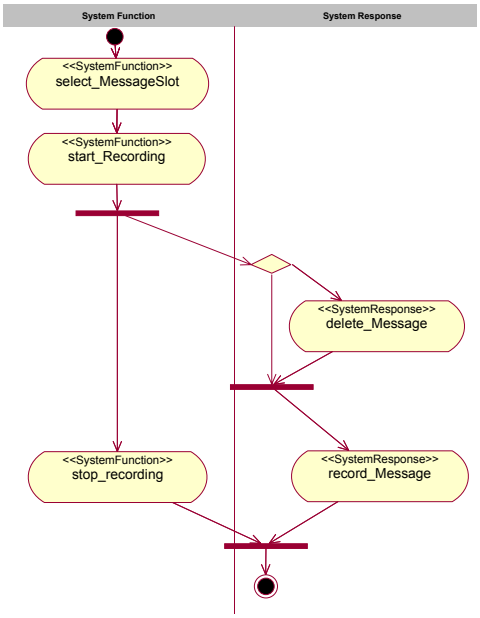

Abbildung 3: Darstellung des Use Case als UML Activity Diagramm

Aus dem so entstandenen Activity-Diagramm lassen sich Testfälle ableiten, indem gemäss festgelegten Überdeckungskriterien Pfade durch den Graphen konstruiert werden. Als Korrektheitskriterium verwenden wir dabei die *Input-Output-Conformance* [Tre96]. Ein- und Ausgaben sind durch Stereotypen gekennzeichnet. Diese können bei der Erstellung von Testfällen genutzt werden. Den abstrakten Systemfunktionen und -reaktionen sind Schnittstellen und Ereignisse zuzuordenen. Für die Systemfunktionen sind implementierungsspezifische Aufrufanweisungen zu erstellen. Für die Systemreaktionen müssen Vergleichsanweisungen zur Überprüfung der tatsächlichen mit den erwarteteten Resultaten festgelegt werden.

Wenn die Testfälle automatisch ausgeführt werden sollen, sind die Aufruf- und Vergleichsanweisungen in ausführbare Routinen umzusetzen. Im Bereich der eingebetteten Systeme können die Aufrufanweisungen aus komplexen Bussignalen bestehen. Die Anweisungen zur Auswertung können den Vergleich kontinuierlicher Signalverläufe erfor-

derlich machen. So wird in unserem Beispiel eine Funktion zum Vergleich der vorgegebenen mit der aufgezeichneten Tonspur benötigt.

4 Weiteres Vorgehen

In diesem Positionspapier haben wir eine Methode zur systematischen Überführung von Use Cases in Test Cases für den automatisierten Systemtest skizziert. Zur Zeit wird diese Methode bei Fraunhofer FIRST prototypisch implementiert und an kommerzielle Werkzeuge angebunden. Weitere Arbeiten bestehen in der Erweiterung der Ausdrucksmächtigkeit der Anwendungsfallbeschreibungssprache, einer Parametrisierung der Methode für verschiedene formale Notationen, sowie einer durchgängigen Toolkette für die kohärente qualitätsgetriebene modellbasierte Entwicklung eingebetteter Systeme.

Literatur

[Coc00] Alistair Cockburn. *Writing Effective Use Cases.* Addison-Wesley, 2000.

[DH01] Werner Damm und David Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[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.

[DPB03] Christian Denger, Barbara Paech und Sebastian Benz. Guidelines - Creating Use Cases for Embedded Systems. IESE-Report Nr. 078.03/E, Fraunhofer IESE, 2003.

[Fri04] Mario Friske. Testfallerzeugung aus Use-Case-Beschreibungen. Präsentation auf dem 21. Treffen der Fachgruppe 2.1.7 Test, Analyse und Verifikation von Software (TAV) der Gesellschaft für Informatik (GI). Softwaretechnik-Trends, Band 24, Heft 3, 2004.

[Har87] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[HVFR04] Jean Hartmann, Marlon Vieira, Herb Foster und Axel Ruder. UML-based Test Generation and Execution. Präsentation auf der TAV21 in Berlin, 2004.

[Mad04] Mike Mader. Designing Tool Support for Use-Case-Driven Test Case Generation. Diplomarbeit, FHTW Berlin, 2004.

[McC03] James R. McCoy. Requirements use case tool (RUT). In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Seiten 104–105. ACM Press, 2003.

[OMG03a] OMG. UML Spezifikation Version 1.5. http://www.omg.org/uml/, 2003.

[OMG03b] OMG. UML Spezifikation Version 2.0. http://www.omg.org/uml/, 2003.

[OMG04] OMG. Model Driven Architecture (MDA). http://www.omg.com/mda/, 2004.

[PL99] Ivan Porres Paltor und Johan Lilius. Digital Sound Recorder - A Case Study on Designing Embedded Systems Using the UML Notation. TUCS Technical Report No. 234, Turku Center for Computer Science, 1999.

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

[RA98] Colette Rolland und Camille B. Achour. Guiding the Construction of Textual Use Case Specifications. *Data Knowledge Engineering*, 25(1-2):125–160, 1998.

[RG99] Johannes Ryser und Martin Glinz. A Scenario-Based Approach to Validating and Testing Software Systems Using Statecharts. In *Proc. 12th International Conference on Software and Systems Engineering and their Applications*, 1999.

[RH04] Matthias Riebisch und Michael Hübner. Refinement and Formalization of Semi-Formal Use Case Descriptions. Position Paper at the 2nd Workshop on Model-Based Development of Computer Based Systems: Appropriateness, Consistency and Integration of Models, ECBS 2004, Brno, Czech Republic, 2004.

[Rup02] Chris Rupp. *Requirements-Engineering und -Management*. Hanser, 2002.

[Sch98] Rolf Schwitter. Kontrolliertes Englisch für Anforderungsspezifikationen. Dissertation, University of Zurich, 1998.

[SHS03] Dirk Seifert, Steffen Helke und Thomas Santen. Test Case Generation for UML Statecharts. In Manfred Broy und Alexandre V. Zamulin, Hrsg., *Perspectives of System Informatics (PSI03)*, Jgg. 2890 of *Lecture Notes in Computer Science*, Seiten 462–468. Springer-Verlag, 2003.

[SL02] Andreas Spillner und Tilo Linz. *Basiswissen Softwaretest*. Dpunkt Verlag, 2002.

[SW01] Geri Schneider und Jason P. Winters. *Applying Use Cases: A Practical Guide*. Object Technology Series. Addison-Wesley, Reading/MA, 2. Auflage, 2001.

[Tre96] J. Tretmans. Test Generation with Inputs, Outputs and Repetitive Quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.

[Win99] Mario Winter. Qualitätssicherung für objektorientierte Software - Anforderungsermittlung und Test gegen die Anforderungsspezifiktion. Dissertation, University of Hagen, 1999.

Towards the Model-Driven Development of Self-Optimizing Mechatronic Systems*

Holger Giese
Software Engineering Group, University of Paderborn
Warburger Str. 100, D-33098 Paderborn, Germany
hg@uni-paderborn.de

Abstract: Advanced mechatronic systems of the future are expected to behave more intelligently than today by building communities of autonomous agents which exploit local and global networking to enhance their behavior and to realize otherwise not possible functionality. While engineering of mechatronic systems and software engineering for embedded systems, multi-agent systems, and distributed systems are established areas, no solution for the systematic development of the outlined future generation of intelligent, distributed, embedded systems exists today. This is not simply a matter of composing the solutions developed for each of these area as some of their requirements are in conflict: E.g., flexibility and autonomy are to some extent at odds with predictability and safety. We propose to address this challenge by a model-driven development approach which includes several advanced analysis and synthesis techniques. A restricted high level UML model serves as a basis for rigorous validation and verification to address the correctness and safety issues. Analyzing the high level models rather than the code is justified by synthesis techniques, which guarantee that all properties of the high level models also hold for the implementation.

1 Introduction

It is expected that in the next generation of mechatronic systems [BSDB00] we will enhance the functionality and improve the performance by exploiting the ever increasing computing resources and available network technology, e.g., in form of wireless networks.

The information processing of these systems is expected to consist of autonomous agents which coordinate with each other and exploit their context knowledge to enhance their behavior. This will be also true for the control aspects of these systems.

To achieve the required intelligent control behavior, we propose to build *self-optimizing* technical systems which may endogenously modify their goals in reaction to changes in the environment.[1] Self-optimizing entities are characterized by their ability to (a) sense their environment and state, (b) adjust their goals accordingly, and (c) adapt their behavior

*This work was developed in the course of the Special Research Initiative 614 - Self-optimizing Concepts and Structures in Mechanical Engineering - University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

[1] http://www.sfb614.de/eng/

to achieve the chosen goals. Such systems thus result in complex real-time coordination as well as sophisticated quasi-continuous control strategies and their context-dependent reconfiguration. To enable the self-optimization of a technical system, rather sophisticated adaptation schemes are desirable at the control level.

Established approaches for the engineering of mechatronic systems and the software engineering of embedded systems, multi-agent systems, and distributed systems exist. However, for the outlined future generation of intelligent, distributed, embedded systems and their mechatronic aspects no simple combination of these approaches can be sufficient. Conflicting goals such as flexibility and autonomy on the one hand and predictability and safety on the other hand as well as the mere complexity which results from the agent interaction renders the development of the future generation of intelligent, distributed, embedded systems a tough challenge.

To address the outlined challenge with a model-driven development approach, we have identified the following ingredients as essential: (1) Appropriate concepts for the modeling with UML for real-time behavior, advanced agent interaction, and the integration of control theory are required. (2) Advanced tools for the analysis of the models w.r.t. correctness and safety are needed in order to justify the model-driven approach in the domain of embedded, safety-critical systems. (3) The efforts spent on the modeling of complete and detailed models is only reasonable when the transition form the abstract model to the implementation is straight forward. Thus, model-driven development must provide sophisticated synthesis algorithms to derive a consistent implementation at a low cost where possible.

In this position paper, we will discuss each of the identified ingredients and then sketch the proposed solution as planed or already realized within the Fujaba Real-Time Tool Suite.[2] We start with modeling concepts for real-time and agents in Section 2. In addition, the concepts for the integration of control engineering and software engineering are presented. After reviewing the required analysis techniques in Section 3, the required synthesis support is discussed (see Section 4). Finally, we summarize the position paper and sketch our proposal for the model-driven development of embedded, safety-critical systems.

2 Modeling

2.1 Real-Time Modeling

The ability to specify required real-time behavior in a clear and unambiguous manner also at the model level is crucial for the development of safety-critical mechatronic systems.

UML Statecharts permit to specify time dependent triggering of transitions with the after and when construct which usually refer to ticks rather than real-time. In addition, transitions are assumed to have zero-execution time which is also at odds with any real-time processing where deadlines and worst-case-execution-times (wcet) have to be considered.

[2] http://www.fujaba.de

The UML Profile for Schedulability, Performance, and Time [OMG02] permits to specify several platform-specific real-time processing attributes for threads or processes such as periods or context switch times, but annotations for behavior description techniques of the platform independent model such as Statecharts are not supported. From the large number of object-oriented modeling approaches for real-time systems ROOM [SGW94] has finally found its way into the UML 2.0 specification [Obj03a]. Inspired by ROOM, UML 2.0 thus supports the specification of the structure of complex systems using components with ports and deployment diagrams; however, specific constructs for the modeling of real-time behavior at a higher level of abstraction are still missing.

Therefore the currently available UML CASE tools at most support soft real-time system development. Rhapsody, Rational Rose/RT, Telelogic Tau, or Artisan Real-time Studio Professional only generate code from Statecharts which realize the logical behavior only, while an appropriate mapping onto threads and scheduling parameters to meet required deadlines in form of the synthesis of a platform specific model remains to be determined in a manual process.

To enable the modeling of real-time behavior within platform independent UML models we propose to extend UML Statecharts with clocks, time guards, and time invariants (cf. timed automata [HMP92]) and equip each transition with a deadline which can be specified relative to the firing time or clocks if required. These *Real-Time Statecharts* [BG03] are supported by the Fujaba Tool and, due to their formal semantics, enable sophisticated analysis of the real-time behavior as well as synthesis which guarantees the real-time deadlines.

On top of this sound foundation, our MECHATRONIC UML approach [BTG04] permits to model complex real-time behavior with component and pattern (cf. [GTB+03]). Besides the components each port/role as well as connectors is equipped with a state machine in form of a Real-Time Statechart to describe the overall real-time behavior for complex systems.

The outlined concepts such as Real-Time Statecharts, patterns, and components are supported by the current version of Fujaba Real-Time.

2.2 Agent Modeling

The multi-agent system paradigm promises to cope with the complexity of the envisioned intelligent mechatronic applications using the agent metaphor. Existing proposals for the modeling of agents with UML are restricted to the abstract information processing level and do not consider real-time or mechatronic aspects.

We propose to achieve the required predictability without ruling out the desired emergent behavior by building the agent modeling concepts on top of the outlined real-time modeling concepts using UML components and patterns. *Communities* are used as the organizational frame to establish behavioral norms for different agents and *Cultures* for each community determine the correct interaction in form of pattern [GBK+03, KG04]. The patterns on the one hand permit to ensure the required safe agent behavior using the

later presented concepts to verify UML patterns. On the other hand the degrees of freedom within the pattern roles permit the agents to interact in an intelligent and autonomous manner.

By explicitly grounding all abstract concepts and rules in the concrete entities of an environment model of the mechatronic system, we can further support formal analysis and rapid prototyping. We further propose to separate the requirements and design into largely independent concerns, realized as social structures with behavioral and communicative norms, and carefully composing them for each agent in such a way that the required analytic properties of one aspect is not invalidated by a second aspect (cf. [KG04]).

Story driven modeling [KNNZ00], as supported by Fujaba already, enables to emulate most of the proposed concepts. However, we plan to add direct support for the modelling of cultures and communities to enable the full potential of the approach.

2.3 Integration of Control

The local information-processing units of self-optimizing mechatronic system have to perform a multitude of functions: control code working in quasi-continuous mode controls motions in the plant, error-analysis software monitors the plant in view of occurring malfunctions, adaptation algorithms adapt the control to altered environmental conditions, planing algorithms determine the long-term goals of the agent, different agents have to be safely coordinated, to name but a few.

Each of these functions shows quite different characteristics. While the control strategies in the controller are usually modeled using CAE tools and block diagrams, the coordination between the units is mainly characterized by reactive and proactive real-time behavior, which is best modeled with real-time variants of state machines. The planing, finally, usually requires flexible and powerful structural and behavioral modeling capabilities as offered by modeling approaches such as the UML. A critical prerequisite to the realization of self-optimizing systems is thus an integration between block diagrams as used in mechanical engineering and the UML as employed by software engineers, both at the conceptual and the tool level.

In context of the UML, a RFP for System Engineering [Obj03b] addresses the general integration problem between control and software engineering domain. However, even the Systems Modeling Language (SysML) [Par04], the only relevant proposal, does only provide a very simple integration of differential equations into the UML which does not support reconfiguration. For ROOM an approach for the integration of control engineering block diagrams into some sort of state machine has been proposed in HyROOM [SPP01]. However, the offered integration permits only to reconfigure block diagrams within a superordinate state machine such that reconfiguration is always restricted to a single component/module.

The overwhelming number of additional functions realized by a single agent makes appropriate structuring techniques imperative when designing the corresponding information-processing unit. Therefore, we propose to use the Operator-Controller-Module (OCM)

[HOG04] architecture. The OCM set-up orientates itself by the kind of effect on the technical system: (1) On the lowest level of the OCM, there is the *controller* featuring an arbitrary number of alternative control strategies. Within the OCM's innermost loop, the currently active control strategy processes measurements and produces control signals. As it directly affects the plant, it is called *motor loop*. The software processing is necessarily quasi-continuous, including smooth switching between the alternative control strategies. (2) The controller is complemented by the *reflective operator*, in which monitoring and controlling routines are executed. The reflective operator operates in a predominantly event-oriented manner. It does not access the actuators of the system directly, but may modify the controller and initiate the switch between control strategies. It furthermore serves as the connecting element to the cognitive level of the OCM. (3) The topmost level of the OCM is occupied by the *cognitive operator*. On this level, the system can gather information concerning itself and its environment and employ it for the improvement of its own behavior.

To realize this architecture, the controller, which can best be modeled using CAE tools and block diagrams, the reflective operator, which can be modeled with real-time variants of state machines, and the cognitive operator, which requires the flexible powerful structural and behavioral modeling capabilities of the full UML, have to be integrated such that the interaction outlined in the architecture can be modeled. The reflective operator may include selected block diagrams which are used for online diagnosis or reactions to certain events. Additionally, the cognitive operator may evaluate a block diagram model in an asynchronously running thread for simulation and prediction purposes.

To support the modular reconfiguration of the internal structures of the controllers, we developed hybrid UML components and a related hybrid Statechart extension for the UML [BGO04]. The hybrid components support the design of self-optimizing mechatronic systems by allowing specification of the necessary flexible reconfiguration of the system as well as of its hybrid subsystems in a modular manner.

An XML Encoding of the hybrid components is currently under development in a student project and a Bachelor thesis to integrate the CASE tool Fujaba Real-Time Tool Suite and the CAE tool CAMeL[3]. Hybrid Statecharts are additionally realized within the student project.

3 Model Analysis

3.1 Safety

A unwanted consequence which results for the outlined trend towards complex interconnected technical systems are serious problems to ensure the system *safety*. Due to the complexity as well as the unpredictable nature of self-optimizing mechatronic systems, applying standard approaches is by no means sufficient any more.

[3] www.ixtronics.de

The current and forthcoming UML versions do not directly support safety-critical system development. Available hazard analysis techniques on the other hand have their origin in the hardware world and do not provide the required degree of integration with software design notations. They assume a very simple hardware-oriented notion of components and therefore do not directly support the identification of common mode faults. Some more advanced approaches [PMRSH01, KLM03, Gru03] support a compositional treatment of failures and their propagation, but still a proper integration with concepts like deployment and the more complex software interface structure is missing.

In [GTS04] our approach for the compositional hazard analysis of the outlined UML models with components and patterns which narrows the described gap between safety-critical system development and available UML techniques is outlined. It builds on the foundation of failure propagation analysis [FMNP94] and permits automatic quantitative analysis at an early design stage. The failures can be modeled as detailed as required using a hierarchical failure classification where correct refinement steps ensure the complete coverage of all possible failures. The approach permits to systematically identify which hazards and failures are most serious, which components or patterns require a more detailed safety analysis, and which restrictions to the failure propagation are assumed. We can thus systematically derive all safety requirements, which correspond to required restrictions of the failure propagation of a single component, pattern, or a system of components and pattern in the UML design.

The presented concepts are currently implemented in Fujaba within a Bachelor and a Master thesis.

3.2 Correctness

The outlined advanced multi-agent systems have, in contrast to classical control systems, rather complex run-time behavior. Therefore, standard means for verification and validation such as testing are by no means sufficient to ensure that the system correctly fulfills the safety requirements identified during hazard analysis.

Knapp et al. present in [KMR02] a tool called HUGO/RT. Within this tool, models are described by UML state machines. The properties to be checked are given as scenarios written as sequence diagrams extended with time annotations. For verification, HUGO/RT transforms the Statecharts into Timed Automata and the sequence diagrams into Observer Timed Automata and applies the model checker Uppaal. The approach of Diethers and Huhn [DH04] is similar to this, but supports the commercial CASE tool (Poseidon).

Another project that aims at modeling and verifying real-time and embedded systems with UML is the OMEGA IST project. The project does not support the complete UML language. Instead, a subset of the UML which is essential for the modeling of industrial real-time applications [DJVP03] is defined. In addition, a subset of the UML is extended by some timing constructs [GOO03] which are necessary when modeling real-time systems. The integrated validation tools support simulation, verification of the properties and automatic test generation.

The applicability of model checking is however rather limited when it comes to the verification of complex distributed embedded real-time systems due to the the state space explosion problem. Only the OMEGA IST project tries to tackle this problem by techniques based on data flow analysis, slicing methods, and simple forms of abstraction, but no compositional model checking approach is provided.

Due to their complexity and history dependent behavior, the addressed complex self-optimizing systems cannot be model checked directly. We therefore propose to ensure their correctness using the composition of the following individual steps: (I) Model checking of Real-Time Statechart including their real-time behavior. (II) Compositional model checking for the distributed coordination of multiple reflective operators. (III) Concepts for the safe integration of the cognitive operator (IV) and finally rules for syntactically checking the correct embedding of hybrid components (controller) into the reflective operator.

(I) As described in [BGHS04], to model check Real-Time Statecharts, we at first map them to HUppaal and then use the tool Vanilla to transform them to timed automata as required by Uppaal [DMY02].

(II) Our approach addresses the state explosion problem for a set of interconnected reflective operators, using compositional model checking and an integrated sequence of design steps (cf. [GTB+03]). These steps prescribe how to compose complex software systems from domain-specific patterns which model a particular part of the system behavior in a well-defined context. The correctness of these patterns can be verified individually because they have only simple communication behavior and have only a fixed number of participating roles. The composition of these patterns to describe the complete component behavior and the overall system behavior is prescribed by a rigorous syntactic definition which guarantees that the verification of component and system behavior can exploit the results of the verification of individual patterns. Compositional model checking and role refinement thus enable the verification of the real-time coordination of large, complex mechatronic systems which result from the interplay of the reflective operators.

(III) To also take the full behavior of the cognitive operators with all their complexity and history dependent evolution into account is not feasible when a complete automatic formal verification is intended. We therefore propose to exploit the architectural separation between the cognitive operator and the reflective operator instead to ensure a safe behavior (cf. [GBK+03]). Thus, the reflective operator filters the input of the cognitive operator to prevent that its unpredictable nature can result in an unsafe operational behavior.

(IV) Self-optimization results in rather complex reconfiguration schemes within the reflective operators and controllers that are composed in a modular manner. We developed an approach [GBSO04] for the modular hierarchical composition of event-based and quasi-continuous behavior where simple consistency checks which are applied for each embedding hybrid components are sufficient to ensure that the reconfiguration only results in correct configurations and that the verified event-based real-time behavior still holds.

The model checking of Real-Time Statecharts, the compositional model checking, and the abstraction from the effects of the cognitive operator are available in the Fujaba Real-Time Tool Suite. The outlined consistency check for the hybrid embedding is planed to be realized within a Master thesis.

4 Advanced Synthesis

During the model-driven development an abstract platform independent model (PIM) is first developed and then refined towards a platform specific one. All properties guaranteed by the PIM have to be preserved by the refinement towards a platform specific model (PSM) and the final code. Thus tool support for the transition from a PIM to a specific PSM which synthesizes required attributes where possible as well as code synthesis for the final target platform is required. Otherwise, ensuring that the high level properties present in the UML models are still present in the implementation becomes in most cases impossible or at least a very tedious task.

The UML Profile for Schedulability, Performance, and Time [OMG02] defines general resource and time models which are used to describe the real-time specific attributes of the modeling elements such as schedulability parameters or quality of service (QoS) characteristics. In terms of MDA, besides a PIM, a more concrete PSM can be specified by using the extensions of the profile. This PSM can be later used for the required model analysis and code generation. However, it remains an open question in the UML profile how all required details of the PSM are determined. In a scenario where the developer maps his model onto the technical concepts such as threads and periods manually, we still have the problem that this mapping results in an iterative manual process of testing and adjusting the model until the real-time constraints are met. Consequently, current CASE tools do not provide sophisticated synthesis of PSM from PIM or code generation which considers resource constraints and guarantees that the real-time constraints are met.

While a number of approaches for synthesis and scheduling analysis for the PSM level exists (e.g., [FGHL04, HSG+01, GKWS03]), there are only a few synthesis approaches which support the developer when refining a PIM towards a PSM. *Modecharts* are a suitable high-level form of state transition systems for the specification of real-time systems, but available code generation does not consider the deadlines or periods [PMS95] . In [ADF+01], scheduling analysis and code generation for timed automata with tasks with WCETs and deadlines associated to locations are presented. However, the presented approach does not take the transition delays into account, arguing that these delays are small compared for the WCETs.

To close the above identified gap between the platform independent model at a high level of abstraction and the platform specific model and the implementation which fulfills the required real-time constraints, we have developed a synthesis algorithm [BGS03]. For a restricted subset of UML and Real-Time Statecharts, the algorithm automatically partitions the model to a PSM and code generation, which take CPU time sharing on a single micro processor into account.

As the PSM and code are synthesized automatically the (platform dependent) WCETs of this implementation are well-known. The automatic partitioning in the PSM respects the WCETs of the local side-effects as well as the WCETs for the implementation of the statechart behavior and the specified deadlines. Therefore the algorithm can guarantee that all real-time requirements are met, which makes an additional analysis unnecessary and avoids a costly iterative manual process. In addition, an integration of quasi-continuous

and event-based discrete models is required to enable reconfiguration. We thus have developed a shared execution framework which supports efficient reconfiguration and modular code generation [BGGO04].

A first version of the outlined high-level code synthesis which includes the partitioning and mapping onto threads has been realized for Java Real-Time. An extension which permits to describe mapping decisions using an explicit platform dependent model and support for C++ is currently under development. The integration of the code execution scheme for UML models and quasi-continuous blocks is currently realized for the non-distributed case in a student project.

5 Summary & Conclusion

The proposed approach addresses the challenge of model-driven development of safety-critical embedded systems as outlined in the introduction by integrating several advanced analysis and synthesis techniques as well as control theory related quasi-continuous approaches with the model-driven development with UML. A high level UML model with some minor extensions serves as a basis for rigorous validation and verification to ensure that required correctness and safety are guaranteed. Synthesis in form of automatic code generation, which guarantees that the all verified properties of the high level model also hold for the implementation justifies that the analysis is done on the more abstract models.

For the outlined model-driven development of self-optimizing mechatronic systems we propose the following process consisting of three main parts (cf. [BTG04]):

In the first part, the safety related requirements are systematically derived using hazard analysis and failure propagation models. Then, individual coordination patterns are developed which realize the non local safety requirements. If it has been successfully verified that the pattern ensure the required safety properties, it is then added to a pattern library.

In the second part, the mechatronic agents are built using the verified coordination patterns stored in the library of patterns by refining and coordinating the pattern roles such that the verified real-time properties are preserved. In the next step further components (e.g. hybrid ones) are embedded into the superordinated component. Simple consistency checks ensure again that the verified real-time properties of the coordination patterns are still valid in spite of the embedding.

As the last part, the PSM and the source code are synthesized for the structure and behavior of the UML model.

The outlined model-driven development is further supported by a sophisticated consistency management subsystem for the integration of different models [BGN+04]. For example, the compositional model checking approach has been realized within Fujaba offering a tight integration for managing the required compositional verification steps using the consistency management subsystem (cf. [BGHS04]) such that an incremental and iterative design and verification process becomes possible.

The presented approach provides the most essential ingredients which are required to engineer the envisioned complex self-optimizing mechatronic systems. Our believe is that the presented solutions address the most crucial problems which have to be resolved to enable the model-driven development of the advanced embedded systems of the future.

Acknowledgements

Thanks to Florian Klein for his comments on an earlier version of this position paper.

References

[ADF+01] Tobias Amnell, Alexandre David, Elena Fersman, M. Oliver Möller Paul Pettersson, and Wany Yi. Tools for Real-Time UML: Formal Verification and Code Synthesis. In *Workshop on Specification, Implementation and Validation of Object-oriented Embedded Systems (SIVOES'2001)*, June 2001.

[BG03] Sven Burmester and Holger Giese. The Fujaba Real-Time Statechart PlugIn. In *Proc. of the Fujaba Days 2003, Kassel, Germany*, October 2003.

[BGGO04] Sven Burmester, Holger Giese, Alfonso Gambuzza, and Oliver Oberschelp. Partitioning and Modular Code Synthesis for Reconfigurable Mechatronic Software Components. In *Proc. of European Simulation and Modelling Conference (ESMc'2004), Paris, France*, October 2004.

[BGHS04] Sven Burmester, Holger Giese, Martin Hirsch, and Daniela Schilling. Incremental Design and Formal Verification with UML/RT in the FUJABA Real-Time Tool Suite. In *Proceedings of the International Workshop on Specification and Validation of UML Models for Real Time and Embedded Systems, SVERTS2004, Satellite Event of the 7th International Conference on the Unified Modeling Language, UML2004*, October 2004.

[BGN+04] S. Burmester, H. Giese, J. Niere, M. Tichy, J.P. Wadsack, R. Wagner, L. Wendehals, and A. Zündorf. Tool Integration at the Meta-Model Level within the FUJABA Tool Suite. *International Journal on Software Tools for Technology Transfer (STTT)*, 2004. (accepted).

[BGO04] Sven Burmester, Holger Giese, and Oliver Oberschelp. Hybrid UML Components for the Design of Complex Self-optimizing Mechatronic Systems. In *Proc. of 1st International Conference on Informatics in Control, Automation and Robotics (ICINCO 2004), Setubal, Portugal*. IEEE, August 2004.

[BGS03] S. Burmester, H. Giese, and W. Schäfer. Code Generation for Hard Real-time Systems from Real-time Statecharts. Technical Report tr-ri-03-244, University of Paderborn, Paderborn, Germany, October 2003.

[BSDB00] David Bradley, Derek Seward, David Dawson, and Stuart Burge. *Mechatronics*. Stanley Thornes, 2000.

[BTG04] Sven Burmester, Matthias Tichy, and Holger Giese. Modeling Reconfigurable Mechatronic Systems with Mechatronic UML. In *Proc. of Model Driven Architecture: Foundations and Applications (MDAFA 2004), Linköping, Sweden*, June 2004.

[DH04] Karsten Diethers and Michaela Huhn. Vooduu: verification of Object-Oriented Designs Using UPPAAL. In Kurt Jensen and Andreas Podelski, editors, *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS2004*, volume 2988 of *Lecture Notes in Computer Science*, pages 139–143. Springer Verlag, 2004.

[DJVP03] W. Damm, B. Josko, A. Votintseva, and A. Pnueli. A Formal Semantics for a UML Kernel Language. Technical Report IST/33522/WP 1.1/D1.1.2-Part1, OMEGA: Correct Development of Real-Time Embedded Systems IST-2001-33522, 2003. Version 1.2.

[DMY02] Alexandre David, Oliver Möller, and Wang Yi. Formal Verification of UML Statecharts with Real-Time Extensions. In Ralf-Detler Kutsche and Herbert Weber, editors, *Proceedings of 5th International Conference on Fundamental Approaches to Software Engineering (FASE 2002)*, number 2306 in Lecture Notes in Computer Science, pages 218–232, Grenoble, France, 2002. Springer Verlag.

[FGHL04] Peter H. Feiler, David P. Gluch, John J. Hudak, and Bruce A. Lewis. Embedded Systems Architecture Analysis Using SAE AADL. Technical Report CMU/SEI-2004-TN-005, Carnegie Mellon University, June 2004.

[FMNP94] P. Fenelon, J. A. McDermid, M. Nicolson, and D. J. Pumfrey. Towards integrated safety analysis and design. *ACM SIGAPP Applied Computing Review*, 2(1):21–32, 1994.

[GBK+03] Holger Giese, Sven Burmester, Florian Klein, Daniela Schilling, and Matthias Tichy. Multi-Agent System Design for Safety-Critical Self-Optimizing Mechatronic Systems with UML. In *OOPSLA 2003 - Second International Workshop on Agent-Oriented Methodologies, Anaheim, CA, USA*, pages 21–32, October 2003.

[GBSO04] Holger Giese, Sven Burmester, Wilhelm Schäfer, and Oliver Oberschelp. Modular Design and Verification of Component-Based Mechatronic Systems with Online-Reconfiguration. In *Proc. of 12th ACM SIGSOFT Foundations of Software Engineering 2004 (FSE 2004), Newport Beach, USA*. ACM, November 2004.

[GKWS03] Zonghua Gu, Sharath Kodase, Shige Wang, and Kang G. Shin. A Model-Based Approach to System-Level Dependency and Real-Time Analysis of Embedded Software. In *The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, Toronto, Canada*, May 2003.

[GOO03] Susanne Graf, Ilena Ober, and Iulian Ober. Timed Annotations with UML. In *Workshop on Specification and Validation of UML models for Real Time and Embedded Systems (SVERTS2003), a satellite event of UML 2003, San Francisco, October 2003*, 2003.

[Gru03] Lars Grunske. Annotation of Component Specifications with Modular Analysis Models for Safety Properties. In Sven Overhage and Klaus Turowski, editors, *Proc. of the 1st Int. Workshop on Component Engineering Methodology, Erfurt, Germany*, 2003.

[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.

[GTS04] Holger Giese, Matthias Tichy, and Daniela Schilling. Compositional Hazard Analysis of UML Components and Deployment Models. In *Proc. of the 23rd International Conference on Computer Safety, Reliability and Security (SAFECOMP), Potsdam, Germany*, Lecture Notes in Computer Science. Springer Verlag, September 2004.

[HMP92] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. What Good Are Digital Clocks? In *9th International Colloquium on Automata, Languages, and Programming (ICALP '92)*, volume 623 of *Lecture Notes in Computer Science*, pages 545–558. Springer Verlag, 1992.

[HOG04] Thorsten Hestermeyer, Oliver Oberschelp, and Holger Giese. Structured Information Processing For Self-optimizing Mechatronic Systems. In *Proc. of 1st International Conference on Informatics in Control, Automation and Robotics (ICINCO 2004), Setubal, Portugal*. IEEE, August 2004.

[HSG+01] Pao-Ann Hsiung, Feng-Shi Su, Chuen-Hau Gao, Shu-Yu Cheng, and Yu-Ming Chang. Verifiable Embedded Real-Time Application Framework. In *Seventh Real-Time Technology and Applications Symposium (RTAS '01), Taipei, Taiwan*, 2001.

[KG04] Florian Klein and Holger Giese. Separation of concerns for mechatronic multi-agent systems through dynamic communities. In Ricardo Choren, Alessandro Garcia, Carlos Lucena, and Alexander Romanovsky, editors, *Software Engineering for Multi-Agent Systems III: Research Issues and Practical Applications*, LNCS. Springer Verlag, December 2004. (to appear).

[KLM03] Bernhard Kaiser, Peter Liggesmeyer, and Oliver Maeckel. A New Component Concept for Fault Trees. In *Proceedings of the 8th National Workshop on Safety Critical Systems and Software (SCS 2003), Canberra, Australia. 9-10th October 2003*, volume 33 of *Research and Practice in Information Technology*, 2003.

[KMR02] A. Knapp, S. Merz, and C. Rauh. *Model Checking timed UML State Machines and Collaborations*. 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT 2002), Oldenburg, September 2002, Lecture Notes in Computer Science volume 2469 pages 395-414. Springer-Verlag, 2002.

[KNNZ00] H.J. Köhler, U. Nickel, J. Niere, and A. Zündorf. Integrating UML Diagrams for Production Control Systems. In *Proc. of the* 22^{nd} *International Conference on Software Engineering (ICSE), Limerick, Irland*, pages 241–251. ACM Press, 2000.

[Obj03a] Object Management Group. *UML 2.0 Superstructure Specification*, 2003. Document ptc/03-08-02.

[Obj03b] Object Management Group. *UML for System Engineering Request for Proposal, ad/03-03-41*, March 2003.

[OMG02] OMG. UML Profile for Schedulability, Performance, and Time Specification. OMG Document ptc/02-03-02, September 2002.

[Par04] SySML Partners. Systems Modeling Language: SysML. Technical report, Aug 2004.

[PMRSH01] Y. Papadopoulos, J. McDermid, b R. Sasse, and G. Heiner. Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure. *Reliability Engineering & System Safety*, 71:229–247, March 2001.

[PMS95] C. Puchol, A.K. Mok, and D.A. Stuart. Compiling Modechart Specifications. In *16th IEEE Real-Time Systems Symposium (RTSS '95), Pisa, Italy*, December 1995.

[SGW94] Bran Selic, Garth Gullekson, and Paul Ward. *Real-Time Object-Oriented Modeling*. John Wiley and Sons, Inc., 1994.

[SPP01] T. Stauner, A. Pretschner, and I. Péter. Approaching a Discrete-Continuous UML: Tool Support and Formalization. In *Proc. UML'2001 workshop on Practical UML-Based Rigorous Development Methods – Countering or Integrating the eXtremists*, pages 242–257, Toronto, Canada, October 2001.

Model-Driven Development of Component Infrastructures for Embedded Systems

Markus Völter
voelter – ingenieurbüro für softwaretechnologie
Ziegelaecker 11, 89520 Heidenheim, Germany
Tel. +49 7321 97 33 44
voelter@acm.org

Abstract: Component infrastructures such as Enterprise JavaBeans, Micorosoft's COM+ and CORBA Components have become a de-facto standard for enterprise applications. Reasons for this success are the clean separation of technical and functional concerns, COTS containers (applications servers), and the resulting well-defined programming model and standardization. To benefit from these advantages in the domain of embedded systems, the same concepts can be used, but a different implementation strategy is required: monolithic application servers are not suitable because of the limited resources regarding computing power, memory, etc. on the device. An alternative can be based on using a family of code-generated containers. The container is generated from models that specify interfaces, components, system topologies and deployments. In addition to motivating the problem and looking at related work, this paper gives general guidelines for the design and implementation of such infrastructures and describes a prototype implementation that has been implemented recently. We also look at the advantages of using such an approach for the electronic control units in vehicles and the benefits the approach could have with regards to vehicle diagnostics.

1 INTRODUCTION

Embedded Software Requirements. Embedded software typically faces some unique constraints not found in desktop software or enterprise systems. These include limited resource, real-time requirements, hardware integration, increased reliability, as well as unit-based cost structures. Based on the requirements discussed in the previous section, we can say that embedded software needs to be more reliable than many other kinds of software, it needs to optimize it's computations for speed and resource consumption, the code size must be minimized and in many cases, real-time requirements need to be verified (maybe empirically) before the software is deployed.

State of the Art. Because of these special requirements, a lot of software for embedded devices is still developed manually, from scratch for each new project. Large-scale reuse is not applied because of the requirement to optimize each piece of software for its particular environment. As a consequence, many techniques that are used to good effect in non-embedded development are not widely used in embedded systems development. Examples are object-orientation, frameworks or reflection. COTS middleware (such as minimum CORBA [21]) is only recently starting to spread in the embedded community. There are several typical high-level application architectures for embedded systems:

- For either very simple of very constrained systems, application code is written directly for the hardware of the device. No operating system is used, some reusable libraries are typically employed, however.

- To allow for some degree of portability of these applications, sometimes a thin abstraction layer is used between the application and the hardware. This can be seen as a simple, custom-developed operating system. Porting the abstraction layer to another device allows for some limited reuse.
- More complex applications typically use more or less powerful realtime operating systems such as VxWorks [31], QNX Neutrino [24] or Osek [26]. Depending on the specific operating system, it handles tasks such as threading, scheduling, device comminication, a file system, etc.
- The most sophisticated application architectures use an OS abstraction layer on top of the operating system to be able to exchange the operating system while not having to rewrite the application code. Figure 1 shows this last alternative.

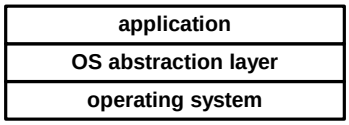

Illustration 1: Illustration 1: High-Level Application Architecture for Embedded Systems

Independent of this structure, parts of the application are generated from models such as state charts or signal flow diagrams and need not be implemented manually. See the related work section for a more detailed discussion.

Component infrastructures [30] provide a (potentially distributed) execution environment for software components. The execution environment is typically called a component container, or container, for short. Components cannot be executed standalone, they require the container to provide essential services. These services handle the technical concerns of an application. Technical concerns are typically cross-cutting aspects that are not directly related to the application functionality implemented with the components. What exactly consititues techical concerns depends on the application domain. In an enterprise environment, the technical concerns are things such as transaction management, resource access decision, fail-over, replication and persistence. The benefit of a component-based approach is that the component (i.e. application) developer does not need to implement the technical concerns over and over again. The developer only *specifies* the container services required by a component, and the container makes sure these services are available to the deployed components. Containers are implemented against some kind of standard (such as EJB [28]) by professional container vendors. Applications just use the containers as they are. Application developers thus don't need to be experts with respect to the (typically non-trivial) technical concerns. The following paragraph lists essential building blocks for component infrastructures. For a more detailed explanation see [30].

Benefits and Liabilities of Component Infrastructures. Using component infrastructures provides several benefits:

- *Portability:* Components are developed against the interfaces of the container, the container can adapt this to different environments (such as operating systems, databases or transaction monitors in the enterprise world).

- *Potential for Container-based Opimization:* Within the boundaries specified by the specifications of the container and the lifecycle interface, the container is free to optimize different aspects of the application.
- *Standardized, Simplified Programming Model:* Because the environment in which components execute is well-defined, and because the developer does not need to deal with low-level implementation details of the technical concerns, the programming model for application devleopers is simplfied and consistent over the family of applications implemented for the same container.
- *Clearly defined developer roles:* Because application developers can focus on their specific application requirements, and because infrastructure experts deal with the implementation of the container, both aspects can be implemented by people who are experts on their respective field, improving the quality of the the software.

Of course there is no such thing as a silver bullet. Typical component infrastructures also suffer from some liabilities. Note that none of these liabilities are inherent to the approach taken by component infrastructures, however, they can be observed in all of today's mainstream implementations:

- *Performance Overhead:* Because requests are intercepted by the container, and because its services are implemented generically to be reusable, performance of component-based applications is impacted.
- *Loss of control:* Some people feel that handing over control over technical aspects to the container limits their control over what is actually happening. While this is true, in most scenarios this is not a liability, however, because the container can handle most of these aspects better and more reliably than code handcrafted by the average developer.
- *Large and heavy:* Most of today's implementations are large and heavy software monsters. Installing, configuring or (re-) starting them can take a while.
- *Complexity:* Of course, by providing a reusable solution to a recurring problem, component infrastructures imply a lot of accidental complexity. This might be a problem for safety-critical applications.

This paper proposes that the benefits presented above would also be desirable in the embedded software world, while ideally not showing the same liabilites. Sections 2 and 3 describes an approach how this could work.

2 PROPOSED SOLUTION

In this paper we propose a component infrastructure that uses model-driven code-generation [12] instead of a generic container. In this context it is critical to understand that we do *not* propose to code-generate the components, i.e. the core application logic. Several tools exists (see related work and [10], [13]) that can generate source code from state charts or signal flow diagrams, and wrapping such functionality in a component is simple. Instead we propose to generate the complete infrastructure that is needed to execute the components on an embedded device, aka the container. The following features are required for a component container for embedded systems:

- *Portable:* Components that are written for a specific container must be able to run on every (real-time) operating system for which a container implementation is available. This requires an abstraction of operating system features[1].
- *Modular:* Enterprise containers typically ship as a big monlithic application that is capable of handling all features of the respective specification, such as transactions, security and persistence. In the embedded world it is not acceptable to carry "excess baggage" in case some features are not needed in a particular application scenario. Consequently, the container infrastructure must be modular itself, only including those features in a particular container instance that are really needed and supported by the target device.
- *Simple:* Again in contrast to the well-known component containers such as EJB, CCM or COM+, a container infrastructure for embedded systems must be lightweight, providing a really simple programming model. Because the target systems (devices) are much more diverse than in enterprise systems, we should focus on the reusable core.
- *Deterministic:* For many embedded applications, determinsm is a critical property. Determinism means that we know in advance (i.e. before runtime) how long something (an operation, a statement) takes to execute. Using dynamic featues such as polymorphism, reflection, etc. makes this kind of determinism much harder to achieve.

Basic Design Decisions. Before we actually look into the implementation of the prototoype, let's look at a couple of additional design decisions that have influenced the system concept, and the prototype described in 3.

First of all, we assume that a system configuration (i.e. the set of components running in a container in the context of an application) is determined statically, before it executes. This is typical for many, but not all embedded applications. So, when the container is generated, the generator knows which components need to run in the container and it knows their resource requirements. As a consequence, the container can validate large parts of the system before it actually starts up. It can detect if a component wants to talk to another component that isn't there, or if a component requires services from the container that cannot be provided because of limitations of the device.

The resulting absence of dynamic decisions has one very big benefit: We are able to statically analyse the code for resource problems or scheduling problems, and with regards to performance and timing using standard code analysis tools [5]. This would not be possible if decisions are taken at run- or load-time. This is the determinism property described in section 2. It is not important for all kinds of embedded systems, but it is important for many.

Second, we assume that the components themselves are written manually. This means that the container generator does not care about the implementation of the components. While there might be some "stub generation" from more abstract interface specifications (such as IDL or a UML model), the component implementation is provided by the application programmer.

---

[1] Portability here does does not neccessarily mean programming-language independence. This is so because the application logic is implemented in a specific programming language, and also the templates (see later) are implemented in a specific language. Of course, the concepts introduced below are independent of any particular programming language, but implementations are not.

Of course, the developer is free to include code in the components that has been generated by state chart or signal flow tools (such as [10], [13]).The following illustration shows the approach described in this paper in a nutshell.

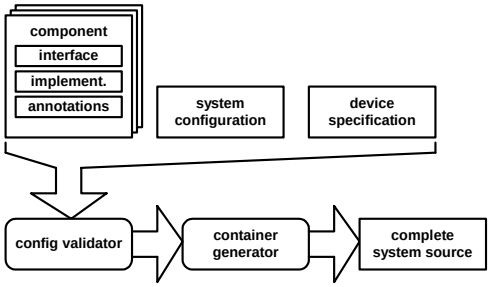

Illustration 2: The proposed soltution in a nutshell

Let's look at the different components in detail. *Components*, as described several times now, contain the application logic, the functional aspect of an application system. The building blocks that constitute a component are the component interface, the implementation and the annotations which state the requirements of the component regarding the container and other components. The *system configuration* specifies which instances are needed of which components, how their resource requirements will be satisfied ("wiring" the components) as well as the configuration of container services and how they apply to components and their instances. The *device specification* describes the available features of the target device (and operating system) on which the resulting application should be deployed. This ultimately determines which container features are available in the target system, as well as how these features are implemented.

All these artifacts are supplied to the *configuration validator* (or buildability checker) that checks if the container will be able to work correctly (as far as this is possible at this early stage). If it determines it is, the artifacts will be supplied to the *container generator* which generates the source code for the container according to the system configuration taking into account the specification in the annotations and the device configuration. Otherwise the container is not code generated and thus cannot be deployed on the device (which is good, because it would not work correctly.)

If the container was generated correctly, this code is then compiled with a normal programming language compiler for the respective target. Optionally, it can be analysed statically to verify its correctness (as far as such static code analysis is feasible [5] – this is no different than static analysis of hand-written code).

Technical Concerns. The selection of what constitutes the technical concerns in a particular family of applications depends on the specific requirements of the domain. Unlike in enterprise systems, where all applications typically consist of some database/transaction related logic, embedded systems as more diverse and it is thus not feasible to decide once and forall on what constitutes the technical concerns. This is the reason why we do not propose one specific embedded container in this paper, but rather an approach, or an architecture, to construct such containers for a specific software system family.

However, there are some candidate aspects that lend themselves to being implemented as container features. Among them are: scheduling, interrupt handling, event propagation, timer, remote communication, generic driver interface, lifecycle control, resource management, safety watchdogs, and advanced error detection.

Interface Specifications. Interfaces play the central role in component infrastructures. Interfaces define contracts among components, and between the container and the components. In traditional systems, interfaces are typically defined as a set of operations including typed arguments, as well as a return type. For serious system composition, more detail must be given on interfaces, including services required from the container to allow the component to run, other component's interfaces required by a component, timing constraints regarding interface operations, pre- and postconditions for operations, or a state machine that defines legal invocation sequences, and data published by a component, or data consumed (required) by a component

Interface definitions as outlined above are logical definitions of what a components provides, or requires. It does not say anything about how these interfaces are implemented. The realization of the interfaces can be supported by the generated container. For example,

- operations can be called directly if the caller and the callee are colocated in the same process, or can include proxies and some kind of remoting infrastructure for remote calls.
- published or required data items can be stored to/retrieved from a shared memory area or it can be put on/taken from a CAN bus.
- timing constraints or pre/postconditions can be checked by the container and errors can be reported

Applicability of the Solution. Considering the different architectures for embedded systems as explained in section 1 the question is: in which architecture can the proposed approach be used sensibly? Let's look at each of these architectures in turn.

- *No operating system:* In these very small systems, the proposed architecture is very suitable. First of all, software on these devices typically is very static, not featuring dynamic aspects. Efficiency and small code size is important, while we still need some flexibility regarding different hardware platforms/devices (because there is no OS). Also, because there is no OS, there is a lot of use for reusable, cross-cutting technical concerns handling of the container. The container thus serves as an efficient implementation of the abstraction layer described in the second architectural alternative – providing flexibility while still being efficient.
- *With (realtime) operating system:* realtime operating systems (as any operating system) typically provide APIs on a very low level. Also, there is no handling of domain- (or software system familiy-) specific technical concerns. Containers can provide this higher-level abstractions. The container can also serve as a means of integrating different tools, systems, middlewares, etc. For example, the container can provide remoting based on CORBA or a different middleware.

3 THE PROTOTYPE

Example Domain. The prototype of a generative component infrastructure for embedded system is currently being developed in the context of automotive ECUs, the electronic control units (i.e., computers and controllers) that control various features of a modern car, such as engine, gearbox, air conditioning, the brake system or the dashboard. A modern middle-class vehicle has about thirty ECUs installed, constituting a distributed system typically based on a CAN network [6] or proprietary topologies. There are several reasons why the software structure of ECUs needs to be standardized and enhanced, for example using a component infrastructure, in addition to the reasons given in section 1:

- The ECUs of different vendors need to interoperate in the context of a vehicle. A coherent software infrastructure is thus necessary.
- The separation of application logic and technical infrastructure as explained in section 2 is especially important, since the same application logic (e.g. brake control) should be reusable in the context of several vehicles, potentially featuring different technical infrastructures. The container can adapt for this.
- Configurability is another important aspect. You want to be able to run the same piece of functionality on different ECUs depending on the vehicle model – you want to utilize the available ECUs as good as possible. A graphical configuration tool that helps is distributing the components to containers and devices.
- The container can also implement a global vehicle state manager (ignition on/off, engine on/off). For example, you are not allowed to reflash (i.e. reprogram) an ECU while the vehicle is driving. As the container can intercept all interactions among components, it is easy for it to track global state and either notify components of state changes or prohibit certain interactions that are not currently allowed.
- Last but not least, diagnosability is a serious issue. After the vehicle has been delivered to the customer, it must be possible to diagnose problems in garages. Typically, an external diagnosis tool is attached to the vehicle. The tool reads the ECUs' internal error buffers and reasons on these errors with the goal of finding the root cause of the problem. Making these tools more efficient and accurate is one of the most urgent tasks for today's after-sales operations. As a precondition, the errors reported by the vehicle must be correct, expressive and accessible through a standardized interface. Also, the description of the ECU topology of the car (which is currently kept outside of the cars in the tool) must be consistent with the actual network deployed in the car. Providing the information based on a reflection on the component infrastructure, can help to avoid inconsistencies. Also, error conditions can be specified abstractly as part of the component definition (such as "raise XYZ error when speed < 100 and fuelLevel > 10"). Code can be generated that efficiently implements the detection of this error on a specific platform.

Prototype Implementation and Technologies. The prototype is implemented in C/C++. The following illustration shows how the prototype is implemented in general.

In the first step, interfaces and (potentially) complex data types are modeled (this can be done using UML 2.0, or using other DSLs). In a second step, we define the components including the interfaces they provide, and the interfaces they require. From these two models, component base code (header files) can be generated; also, complex type implementations and interfaces are created. In the next step (step 4), the implementation of those components can be created manually by the developers. This completes the first phase, *component development*.

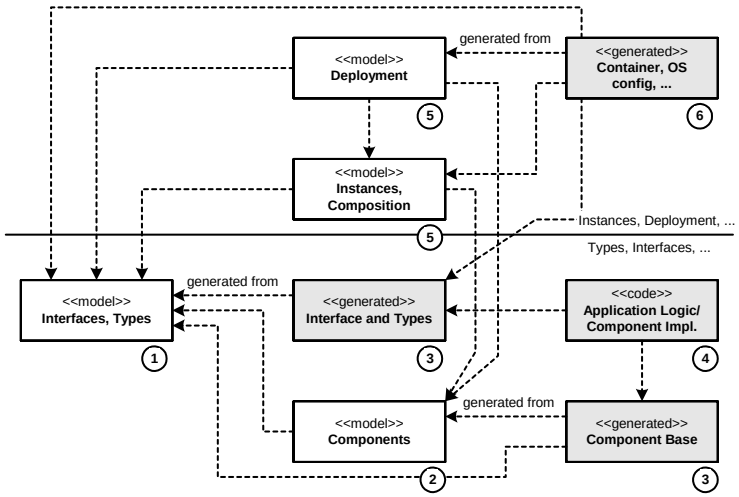

Illustration 3: Prototype implementation structure

In a second phase, *system development*, we define component instances and the connections among those instances. Then we specify the deployment of these instances on hardware elements of the system (those specifications are not shown in illustration 3). Based on these models, we can generate the containers for the hardware elements, as well as the OS config files.

In the prototype, we generate the code using the openArchitectureWare generator framework [4]. This particular generator tool is based on an explicitly programmed metamodel which is implemented in Java. Thus, the generator needs to be supplied with the metaclasses that describe the metamodel for the various models used in the approach. Also, we need to define templates that specify the mapping from the metamodel to the generated source code

The output of the generator is the source code (skeletons) for the components, the complete container implementations, as well as a *make* file used to compile and build the container and the components. The subsequent C++ compiler/linker must be fed with these generated sources, the manually created component implementation files as well as additional runtime libraries. We also generate suitable config files for the operating system, OSEK in our example.

The configuration file itself can be set up using a graphical configuration tool based on the Eclipse framework. It imports the interface definitions from the models and allows the configuration of instances, their threading behaviour, association of instances to containers, automatic remoting, event propagation, etc.

Prototype features. This section focuses on some noteworthy features implemented in the prototype.

Application functionality is realized by having components collaborate – a component instance invokes operations on other components instances. It is important that such invocations implies only the smallest overhead possible. For example, if the container does not need to intercept invocations (because the configured technical concerns don't require intervention by the container), an operation invocation does not have *any* overhead at all. An ordinary method invocation is used. If, for example, a method should be invoked asynchronously (which needs to be specified at generation time), then the container generator generates a proxy for the instance. The proxy, when an operation is invoked, creates a thread (or obtaines one from a pool) and then subsequently invokes the operation on the instance in this thread. Whenever a client (component) wants to get a reference to the instance, the container makes sure that the proxy is returned instead of the real instance. Consequently, operations on the instance are invoked asynchronously without any involvement of the client or the component implementation.

The same conceptual approach is taken when an instance invokes operations on a remote component instance. The client component's container contains a proxy that translates the call to whatever remoting technology is configured – CORBA, sockets, or something else. In the server container, there is another proxy that receives the remote message and invokes the target operation on the target instance. Both proxies are automatically generated based on information in the configuration file.

Signals are simple noifications (typically integers) that are exchanged among component instances. The propagation of signals is handled by the container. The configuration file specifies which signals should be propagated to which component instance. If a component raises a signal, the container propagates the signal to all receivers. If the configuration file specifies that the propagation should happen asynchronously, the container creates a thread and handles propagation in this thread.

Note that if there are no signals to be propagated, the generated container does not contain any propagation logic, i.e. no runtime overhead and no size overhead.

Last but not least let's have a look at the diagnosis-specific extensions for the tool.

First of all, a generic diagnostic interface is defined in the model. All components are required to implement this interface for generic access by an external diagnosis tool and to allow components to query other components for their state (à la "if I have a problem, let's see if my supplier also has a problem which might cause my own problem"). The code generator is later supplied with the DTC/FaultCode specification (a specification that defines which errors might occur in an ECU and how the ECU can detect them) so that the implementation for the diagnostic interface can be generated to a large extent. Note that this very same specification, together with the config file (which specifies the topology and the dependencies) can then be supplied to external diagnosis tools, which uses this information as the basis for its diagnoses.

The generator also receives the state/timing information for the component interfaces. The generator generates code into the container that diagnoses errors in the timing/state sequence of components efficiently and reports them. Finally, the implementation of the application logic (e.g. controlling the anti skid system) can be generated from other tools such as Matlab [13] or Statemate [10], if necessary.

4 RELATED WORK

Infrastructures for embedded systems. Several efforts are currently undertaken regarding the provision of infrastructure for embedded software development. Let's look at some of them.

OSGi, the Open Services Gateway Initiative [23] aims at providing infrastructure for dynamic service infrastructures on devices, so-called gateways. Gateways are considered to be "facades" around complex distributed, embedded systems (such as vehicles, wired homes, industrial estates) onto which services can be installed remotely. OSGi implementations help in installing these services, tracking dependencies among them, starting and stopping services, etc. In addition to this basic functionality, OSGi provides a set of services, e.g. an simple HTTP server, messaging or a generic driver interface for hardware. In contrast to the approach proposed here, there is no notion of a container as such, because the OSGi infrastructure does not handle crosscutting technical concerns for the installed services. It only serves as a framework that handles some, well-defined tasks. Also, OSGi targets dynamic environments where services can be dynamically installed and removed at runtime. This is in direct contrast to the approach presented in this paper, where as much as possible is generated statically. As such, this approach is targetted at the core embedded system whereas OSGi systems are targetted for dynamic gateways.

There are several implementations of **CORBA** [16] for embedded devices. The TAO ORB [7] can be used in embedded settings, it is available for realtime embedded operating systems such as QNX [24] and it is currently ported to really small, embedded OSeK environments [26]. CORBA, however, does not provide a component infrastructure. CORBA, especially the embedded versions based on the minimum-CORBA specification [21], provides a means allow remote operation invocations, not very much more. As such it can serve as a basis for some of the features provided by the container proposed in this paper. The CORBA component model [16], which does provide a container/component infrastructure on top of CORBA is a very sophisticated component infrastructure that is much too complicated for the embedded world. Also, no implementations are currently available.

Lightweight component containers. Most current implementations of component containers (specifically for EJB) are rather large, monolithic tools and neither intended nor suitable for embedded systems. However, a couple of projects aim at creating smaller, more modular component containers. For example the JBoss EJB implementation [11] has the concept of the "generalized aspect container". A container for components can be configured with an arbitrary set of interceptors that can each handle a specific aspect. This is a rather flexible approach, and the functionality of the container can be adapted to the specific needs of the system. However, JBoss uses reflection for all this and thus does not optimize for performance. While this approach is conceptually not far from what I propose in the paper, it is not suitable to use in embedded systems because of its dynamism. Also, it is currently bound to the EJB component model [28] and thus, the Java programming language.

In general, aspect oriented programming [2] is a way to selectively introduce crosscutting (typically technical) concerns into an application. It is thus a good way to build lightweight, modular component containers. A container feature is basically an aspect. AspectJ [9] is a Java AOP extension that can be used for this purpose, specifically as it is based on static code weaving [12]. The Java Aspect Components framework [1] is an attempt at building a generic framework for providing a selection of technical concerns for enterprise applications (failover, persistence, GUI, etc.). It is based on Java and uses mainly reflection and other dynamic techniques. Again, this tool is not explicitly targetted for small embedded environments.

In several vertical domains, standards are currently being defined for component infrastructures. A popular example is the AUTOSAR [35] standard that is currently being defined for the automotive domain.

Modularized Infrastructure. The idea of providing reusable services to applications is not revlolutionary at all. Operating systems do exactly this. Realtime operating systems for use in embedded systems provide a set of services to the applications that run on them. Some realtime and embedded operating systems such as QNX [24], OseK [26] or even Windows CE [15] are even customizable in the sense that the image that is deployed to the embedded device only contains the features required by the particular application. As such it can be seen as some kind of "component container" with the applications being the components. However, there are several important differences: First of all, the developer is not able to extend the infrastructure (i.e. the operating system) with additional technical concerns. In contrast, the approach presented in this paper can be adapted with new container features at any time. Second, operating systems do not do things such as creating proxies for threading or remote access to other programs. Operating system features, especially embedded, realtime OS features, are typically much more low-level.

Code Generation. The approch presented in this paper is based primarily on soure code generation (for an overview of code generation technologies, see [12]). Source code generation is already heavily used in embedded software development in tools such as Statemate [10] or Matlab/Simulink [13]. However, these tools don't use the principle of separation of concerns to factor out and generate the code for handling the technical concerns, instead they typically create the "application logic" from signal flow diagrams or state charts.

These tools can be easily integrated with the approach presented in this paper by "wrapping" the functional code generated by them in components that can be deployed on the container generated by the approach here.

Model Driven Software Development. Model-Driven Software Development (MDSD) is concerned with generating complete applications from models. Those models can be anything that is useful to specify application functionality on an abstraction level higher than implementation code – optionally a domain-specific notation can be used. OMG's MDA [20] is a standard to use UML [22] for model-driven development. The approach presented in this paper uses model-driven techniques extensively. The models are specified in UML and other notations such as XML. While Model-Driven Software Development aims at (but does not require) the generation of the complete application including the behavior, we only generate infrastructure code here. Implementation of the core application logic is out of scope. For more information on MDSD see [32] and [33]

5 PRACTICAL EXPERIENCE

I have been part of several projects implementing component infrastructures for various domains (among others, automotive and mobile phones). Although I cannot provide details about these projects in this paper, the approach has proven *very* successful. Specifically, the tools that are required for the generative aspect of the approach are practically usable and easy to use. MDSD makes the concepts of components and communication middleware as explained in [30] and [33] applicable to the embedded domain. Please contact the author in case you want to know details.

6 ACKNOWLEDGEMENTS

Several people provided valuable feedback on earlier versions of this paper: Frank Buschmann, Michael Englbrecht, Michael Kircher, Alexander Schmid and Uwe Zdun. Many thanks to all of them.

7 ABOUT THE AUTHOR

Markus Völter works as an independent consultant on software engineering and technology, focusing primarily on software architecture, middleware and model-driven software development. He has experiences in many different domains including health care, banking, astronomy, mobile system and automotive; over the last year, Markus has worked in the embedded domain, using a model-driven approach to implement component/container infrastructures in the automotive and mobile phone domains.

In addition, Markus is an active author and conference speaker. He is known as an authority on middleware and model-driven software development and regularly speaks on this topic at conferences such as OOP, JAOO, ECOOP or OOPSLA. Markus is the co-author of *Server Component Patterns* and *Remoting Patterns* pattern books as well as of the forthcoming dPunkt title on *Model-Driven Software Development*. Details can be found at www.voelter.de

8 REFERENCES

[1] AOPSYS, *Java Aspect Components*, http://jac.aopsys.com/

[2] AOSD steering committee, *Aspect-Oriented Software Development*, http://aosd.net

[3] ASN.1 consortium, *ASN.1 home page*, http://www.asn1.org/

[4] Sourceforge.net, *openArchitectureWare,* www.openarchitecturware.org

[5] Chung, T.M.; Dietz, *Static scheduling of hard real-time code with instruction-level timing accuracy*, http://www.computer.org/proceedings/rtcsa/7626/76260203abs.htm

[6] CiA, *Controller Area Network (CAN), an overview*, http://www.can-cia.de/can/

[7] Doug Schmidt, *The ACE ORB*, http://www.cs.wustl.edu/~schmidt/TAO.html

[8] Doug Schmidt, *The Adaptive Computing Environment*, http://www.cs.wustl.edu/~schmidt/ACE.html

[9] Eclipse.org, *The AspectJ project*, http://www.eclipse.org/aspectj/

[10] Ilogix, Inc., *Statemate MAGNUM*, http://www.ilogix.com/products/magnum/index.cfm

[11] JBoss.org, *The JBoss Application Server*, http://www.jboss.org/

[12] Markus Voelter, *A collection of Patterns for Program Generation*, http://www.voelter.de/data/pub/ProgramGeneration.pdf

[13] Mathworks, *Matlab / Simulink*, http://www.mathworks.com/products/tech_computing/

[14] Microsoft, *COM+ Specification*, http://www.microsoft.com/com/tech/COMPlus.asp

[15] Microsoft, *Windows CE*, http://www.microsoft.com/windows/embedded/ce.net/default.asp

[16] OMG, *CORBA*, http://www.corba.org/

[17] OMG, CORBA and the CCM, http://www.corba.org/

[18] OMG, CORBA, XML and XMI® Resource Page, http://www.omg.org/technology/xml/

[19] OMG, Meta-Object Facility (MOF), version 1.4, http://www.omg.org/technology/documents/formal/mof.htm

[20] OMG, Model-Driven Architecture, http://www.omg.org/mda

[21] OMG, Minimum CORBA Specification, http://doc.ece.uci.edu/CORBA/formal/02-08-01.pdf

[22] OMG, UML Resource Page, http://www.omg.org/uml/

[23] OSGi Consortium, The Open Services Gateway Initiative, http://www.osgi.org/

[24] QNX, QNX Neutrino RTOS, http://www.qnx.com/products/ps_neutrino/

[25] Realtime Development Corp, Realtime defined, http://www.realtimeonline.com/RealTimeDefined.htm

[26] Several, OSEK/VDX, http://www.osek-vdx.org/index.htm

[27] Several, Simple Network Managment Protocol, http://www2.rad.com/networks/1995/snmp/snmp.htm

[28] Sun Microsystems, EJB Specification, http://java.sun.com/products/ejb/

[29] Unisys, Unisys website, http://www.unisys.com

[30] Voelter, Schmid, Wolff, Server Component Patterns - Component Infrastructures illustrated with EJB, Wiley, 2002

[31] Windriver Software, VxWorks, http://www.windriver.com/products/vxworks5

[32] Markus Völter: MDSD Tutorial, http://www.voelter.de/services/mdsd-tutorial.html

[33] Voelter, Kircher, Zdun: Remoting Patterns, Wiley 2004

[34] Stahl, Voelter, Bettin: Modellgetriebene Softwareentwicklung, dPunkt 2005

[35] AUTOSAR consortium, AUTOSAR website, http://www.autosar.org

Systemverhaltensmodelle zur Spezifikation bei der modellbasierten Entwicklung von eingebetteter Software im Automobil

Matthias Grochtmann, Linda Schmuhl

Labor Software-Technologie, Methoden und Tools (REI/SM)
DaimlerChrysler AG, Forschung und Technologie
Alt-Moabit 96a
10559 Berlin
Matthias.Grochtmann@daimlerchrysler.com
Linda.Schmuhl@daimlerchrysler.com

Abstract: Bei der Entwicklung eingebetteter Software im Fahrzeug hat sich die modellbasierte Entwicklung durchgesetzt. Typischerweise beginnt in der Praxis die Modellierung mit dem physikalischen Modell, welches die Funktion bereits vollständig realisiert, wobei noch von den realen Einschränkungen im Fahrzeug abstrahiert wird. Implementierungsmodell und Codegenerierung folgen. In diesem Papier werden so genannte Systemverhaltensmodelle als ein Ansatz vorgestellt, früher und anschaulicher mit der Modellierung zu beginnen. Systemverhaltensmodelle werden mit den gleichen Sprachmitteln wie die folgenden Modelle beschrieben, dienen jedoch primär der Spezifikation des Verhaltens der zu realisierenden Funktion und sind keine Realisierung. Sie ersetzen oder ergänzen somit die übliche textuelle Anforderungsspezifikation. Gegenüber einer textuellen Spezifikation bieten sie jedoch verschiedene Vorteile: sie sind beispielsweise simulierbar, sind halbformal und dadurch präziser und erlauben eine ganzheitliche, graphische Spezifikation.

1 Einleitung

Die Funktionen eines Automobils werden zunehmend durch Software und Elektronik bestimmt. Systeme wie ABS, ESP (Electronic Stability Program) oder moderne Motorsteuerungen sind ohne Software nicht mehr realisierbar; die Anforderungen der Kunden und Zulassungsstellen wären ohne Software nicht mehr erfüllbar, beispielsweise sind moderne Abgasnormen und geringe Verbrauchswerte nur durch ein ausgefeiltes Motormanagement erreichbar; ESP senkt die Unfallhäufigkeit drastisch und ist damit für viele Kunden unverzichtbar geworden. Adaptive Tempomaten (Distronic) können den Verkehrsfluß beruhigen und damit Staus vermeiden, sofern ein bestimmter Prozentsatz an Fahrzeugen damit ausgestattet ist usw. Es ist zu erwarten, daß die Bedeutung von Software im Fahrzeug weiter zunehmen wird.

Eine methodische, werkzeuggestützte Softwareentwicklung ist dabei notwendig, um angesichts der Komplexität dennoch die notwendige Qualität der Software bei vertretbarem Entwicklungsaufwand zu erreichen. Tatsächlich ist ein erheblicher Teil der Fahrzeugpannen inzwischen der Elektronik geschuldet.

Bei der Entwicklung von Software für Automobile vollzieht sich seit einigen Jahren ein Paradigmenwechsel. Dieser ist durch einen Übergang von der klassischen Programmentwicklung hin zu modellbasierten Techniken gekennzeichnet, bei denen Modelle eine zentrale Rolle spielen.

2 Modellbasierte Entwicklung

Bei der Entwicklung eingebetteter Software im Fahrzeug hat sich die modellbasierte Entwicklung durchgesetzt [Kl04]: Typischerweise wird nach der Ermittlung textueller Anforderungen mittels aufeinander aufbauender graphischer Funktionsmodelle (insbesondere: physikalisches Modell, welches die Funktion bereits vollständig realisiert, wobei noch von den realen Einschränkungen im Fahrzeug abstrahiert wird; Implementierungsmodell, welches zusätzlich die technischen Randbedingungen berücksichtigt) die gewünschte Funktion realisiert und letztlich aus dem Modell Code generiert, der dann in das Steuergerät integriert wird. Begleitend wird verifiziert und validiert, wobei Tests aufgrund der Ausführbarkeit der Modelle bereits sehr früh erfolgen können. Als Sprachmittel für die Modelle wird häufig Matlab/Simulink/Stateflow verwendet [Ma04].

Vorteile der modellbasierten Entwicklung sind die stets simulierbaren Funktionsmodelle und die automatische Umsetzung in Code (Codegenerierung) sowie die Möglichkeit zum entwicklungsbegleitenden Testen. Eine weitgehend durchgängige Methoden- und Toolkette vereinfacht die Entwicklungsunterstützung. Aus diesen Gründen verspricht die modellbasierte Entwicklung gegenüber einer klassischen Softwareentwicklung Effizienzgewinne.

Jedoch setzt die modellbasierte Entwicklung mit dem physikalischen Modell erst relativ spät ein und umfaßt bereits eine konkrete Realisierung aller Regelungsalgorithmen. Davor wird üblicherweise nur mit textuellen, vereinzelten Anforderungen spezifiziert, zum Beispiel mit dem Werkzeug Doors [Te04], so daß hier ein Bruch in der Durchgängigkeit vorliegt.

Manche Praktiker verzichten sogar gänzlich auf ausgefeilte Anforderungen, um den Bruch so zu vermeiden. Sie argumentieren, daß das physikalische Modell selbst die Spezifikation sei. Auch wenn man formal akzeptieren kann, daß das physikalische Modell die Abbildung von Eingängen nach Ausgängen eindeutig festlegt, ist die Beschreibungsebene viel zu detailliert und von Implementierungsdetails getrieben, als daß diese Sichtweise angemessen wäre.

Eine abstrakte, das WAS und nicht das WIE festlegende Beschreibung ist auch bei der modellbasierten Entwicklung notwendig, insbesondere auch als Referenz für den Test der nachfolgenden Modelle und des Codes. Textuelle Anforderungen sind dabei jedoch nicht die einzige mögliche Lösung.

3 Systemverhaltensmodelle

In diesem Papier werden *Systemverhaltensmodelle* als ein Ansatz vorgestellt, früher und anschaulicher mit der Modellierung zu beginnen. Systemverhaltensmodelle werden mit den gleichen Sprachmitteln wie die folgenden Modelle beschrieben, dienen jedoch primär der Spezifikation des Verhaltens der zu realisierenden Funktion und sind keine Realisierung. Sie ersetzen oder ergänzen somit die übliche textuelle Anforderungsspezifikation. Gegenüber einer textuellen Spezifikation bieten sie jedoch verschiedene Vorteile: sie sind beispielsweise simulierbar, sind halbformal und dadurch präziser und erlauben eine ganzheitliche, graphische Spezifikation.

Viele Fahrzeugsysteme lassen sich erfahrungsgemäß sehr gut mit Zustandsübergangsdiagrammen beschreiben und Zustandsübergangsdiagramme zur Spezifikation sind in der klassischen Informatik bewährt. Deshalb werden als Sprachmittel für die Systemverhaltensmodelle Zustandsübergangsdiagramme in Form von Stateflow-Diagrammen eingesetzt. Die Verwendung von Stateflow, welches neben dem blockorientierten Simulink auch in den physikalischen Modellen und in den Implementierungsmodellen verwendet wird, weitet die Durchgängigkeit der Toolkette auf die Spezifikation aus, erhöht die Akzeptanz bei den Entwicklern und läßt die Wiederverwendung von Teilen der Spezifikation in späteren Phasen zu.

Ein Systemverhaltensmodell ist zum Beispiel nützlich, wenn es bei einem großen, komplexen System vor allem auf die richtige Interaktion von Teilfunktionen ankommt, wenn man viele systeminterne Zustände hat, die verknüpft sind, oder wenn man überprüfen will, ob das Systemverhalten vollständig und konsistent beschrieben ist; letztlich also vor allem dann, wenn es mehr auf die Gesamtsicht ankommt als auf Details. In solchen Fällen ist Graphik oft angemessener als Text. Weiterhin kann das Systemverhaltensmodell auch simulierbar gestaltet werden, so daß man das grundsätzliche Systemverhalten bereits früh erproben kann.

Die Ein- und Ausgaben des Systemverhaltensmodells liegen auf der Ebene von Fahreraktionen, Ausgaben an den Fahrer, Fahrzeugzustand und Umgebung. Das Systemverhaltensmodell soll das generelle Ein- und Ausgabeverhalten sowie die verschiedenen Zustände des Systems und ihr Zusammenspiel beschreiben, jedoch noch nicht die Funktion vollständig realisieren. Folglich sind informelle Anteile, zum Beispiel textuell beschriebene Übergangsaktionen, und Vereinfachungen, zum Beispiel bevorzugt boolesche Ein- und Ausgaben, enthalten. Trotzdem ist es möglich, das Systemverhaltensmodell simulierbar zu gestalten, so daß die oben genannten Aspekte der spezifizierten Funktion erprobt werden können.

Wir haben Systemverhaltensmodelle an Funktionen des Innenraums und an Fahrerassistenzsystemen erprobt: die Ergebnisse sind vielversprechend; die Verhaltensmodelle waren gut lesbar und geeignet, die funktionalen Anteile des Systems anschaulich und präzise darzustellen. Teilweise haben wir Systemverhaltensmodelle parallel zu textuellen Anforderungen erstellt: die Erfahrung dabei war, daß wir das Modell in den meisten Fällen als Hauptdokument verwendet haben, an dem wir die eigentlichen Festlegungen trafen, und daraus die textuellen Anforderungen abgeleitet haben. Im Fall einer Fahrerassistenzfunktion aus dem Einparkbereich haben wir ein Systemverhaltensmodell erfolgreich verwendet, um die ca. 60 textuellen Anforderungen auf Vollständigkeit und Konsistenz zu überprüfen, was aufgrund der Vereinzelung der Anforderungen ohne Systemverhaltensmodell Probleme aufwarf.

Abbildung 1 zeigt die oberste Ebene eines Systemverhaltensmodells für die Regelung eines Scheibenwischers mit Regensensor (SMR)[1]. Eingänge sind Fahrereingaben, wie die Position des Lenkstockschalters, und der Fahrzeugzustand, z.B. ob die Fahrertür geöffnet ist oder der Sensor defekt ist. Prinzipiell könnte auch der Umgebungszustand in das Modell eingehen, z.B. die Regenintensität. Man beachte, daß es sich nicht um die realen Signale, die später in das Steuergerät eingehen werden, handelt, sondern um abstrakte Eingänge auf Anforderungsebene. Die Typen sind daher in diesem Beispiel auf Boole und Aufzählung beschränkt.

Das Systemverhaltensmodell ist zunächst als Dokument gedacht, welches ein Mensch lesen und verstehen soll. Eingebettet in Simulink und stimuliert mit Eingabeverläufen ist es aber mittels der Matlab-Werkzeuge auch simulierbar, so daß zum Beispiel Use Cases durchgespielt werden können. Um bei der Simulation das Verhalten des Modells beobachten zu können, werden als Ausgaben aus dem Modell einerseits Informationen über die jeweiligen Zustände des Modells, andererseits Informationen über die Aktionen, die jeweils ausgeführt würden, herausgeführt.

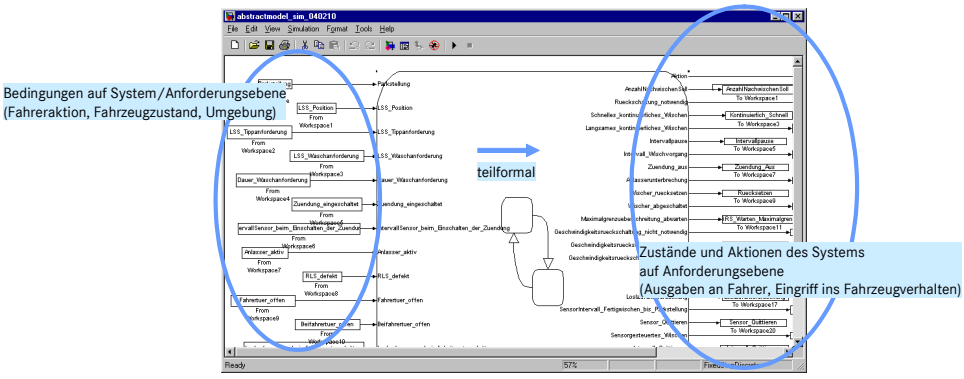

Abbildung 1: Verhaltensmodell des SMR

[1] Diese SMR-Funktion wurde von der DaimlerChrysler AG, Forschung REI/SM angefertigt und dient als Beispiel ohne jeden Serienbezug.

Abbildung 2 zeigt exemplarisch einen Ausschnitt aus dem Systemverhaltensmodell für den SMR. Man sieht, daß Zustände und Übergangsbedingungen formal ausgeführt sind, wodurch das Zustandsübergangsverhalten des Systems exakt beschrieben ist. Die eigentlichen Aktionen des Systems werden dagegen informell beschrieben, um eine einfache Beschreibung zu erlauben und die Realisierung nicht vorwegzunehmen. Um die Simulierbarkeit des Verhaltensmodells sicherzustellen, werden die Aktionen jedoch in Stateflow-Syntax niedergeschrieben. Dabei haben wir zwei Möglichkeiten erprobt: Naheliegend ist es, für jede Aktion eine boolesche Variable einzuführen, deren Name die Aktion beschreibt. Die Variable wird dann jeweils auf true oder false gesetzt. Dies führt allerdings zu einer großen Zahl an Variablen, die zu verwalten sind, und damit auch zu einer großen Zahl von Ausgängen aus dem System. Eleganter ist es, für jede Aktion eine Konstante einzuführen, deren Name wiederum die Aktion beschreibt und deren Wert ein Identifikator (z.B. Nummer) für die Aktion ist. Es gibt dann eine „Aktionsvariable“, der die jeweilige Aktion in Form ihrer Konstante zugewiesen wird. In diesem Fall hat man nur einen Ausgang aus dem Modell für die Aktionen. Allerdings muß man diesen Ansatz erweitern, wenn man Aktionen auch parallel zulassen will, etwa durch parallele Automaten oder über verschiedene Ebenen des Stateflows hinweg: dann sind ggf. mehrere Aktionsvariablen notwendig.

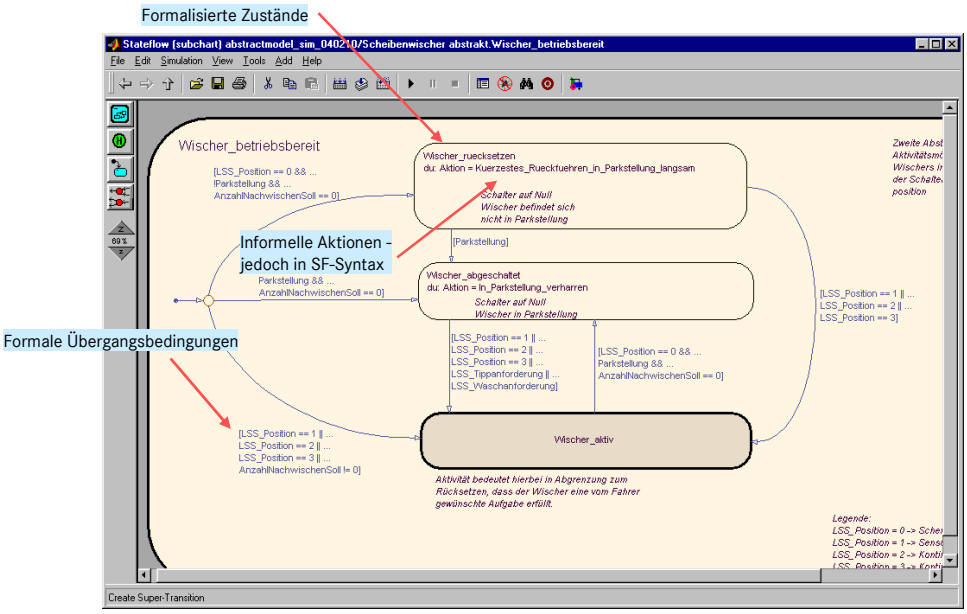

Abbildung 2: Zustände, Übergänge und Aktionen im Verhaltensmodell des SMR

Um die Beschreibung lesbar zu halten, empfiehlt es sich, die vielfältigen Möglichkeiten von Stateflow nicht zu stark auszunutzen. Insbesondere empfehlen wir, parallele Automaten nur dann zu verwenden, wenn auch die Aufgabenstellung Parallelität enthält.

4 Abschluß

Systemverhaltensmodelle können bei der Entwicklung von Software im Fahrzeug helfen, bereits frühzeitig auf der Anforderungsebene mit der Modellierung zu beginnen. Erste Erfahrungen sind positiv. Dennoch sind weitere Diskussionen und Festlegungen zu leisten:

- Inwieweit das Systemverhaltensmodell die textuelle Anforderungsspezifikation nur ergänzen oder teilweise ersetzen sollte, ist zu untersuchen. Modell und Anforderungen dürfen sich natürlich nicht widersprechen; ob Redundanzen sinnvoll sind, ist offen. Stets notwendig ist eine separate Beschreibung für die nicht funktionalen Anteile, wie z.B. Anforderungen an Ressourcen oder zeitliches Verhalten.
- Die beste Form des Modellierens eines Systemverhaltensmodells sollte weiter betrachtet werden. Wie sollten die Eingaben und Ausgaben strukturiert und typisiert sein; welche Sprachelemente sollten Verwendung finden; inwieweit sind doch bereits formalisierte Aktionen hilfreich; wann sind parallele und hierarchische Diagramme sinnvoll; welcher Abstraktionsgrad der Beschreibung ist angemessen?
- Da die Informationen im Systemverhaltensmodell teilweise formalisiert vorliegen, könnte man daraus versuchen, für die weitere Entwicklung hilfreiche Informationen abzuleiten, zum Beispiel Testfälle mittels White-Box-Kriterien.
- Es sind auch Wege möglich, das Systemverhaltensmodell durch geeignete Erweiterung und Einbettung in Richtung auf ein physikalisches Modell zu transformieren. Wie dies genau zu leisten wäre und ob dieser Weg praxistauglich sein könnte, ist zu diskutieren.
- Weitere Anwendungsfelder über die Automobiltechnik hinaus wären zu betrachten und zu erproben.

Die Beantwortung obiger Fragen könnte den Nutzen von Systemverhaltensmodellen bei der Entwicklung von Software in technischen Systemen weiter steigern und damit auch die Qualität und Effizienz der gesamten Entwicklung in diesem Bereich.

Literaturverzeichnis

[Kl04] Klein, T.; Conrad, M.; Fey, I.; Grochtmann, M.: Modellbasierte Entwicklung eingebetteter Fahrzeugsoftware bei DaimlerChrysler. In (Rumpe, B.; Hesse, W. Hrsg.): Proceedings Modellierung 2004, Marburg, 23.-26. März; GI-Edition Lecture Notes in Informatics, S. 31-41

[Ma04] www.mathworks.com

[Te04] www.telelogic.com

OO Model based programming of PLCs

Albert Zündorf, Leif Geiger, Jörg Siedhof

University of Kassel, Software Engineering Research Group,
Department of Computer Science and Electrical Engineering,
Wilhelmshöher Allee 73,
34121 Kassel, Germany
{leif.geiger, albert.zuendorf}@uni-kassel.de
http://www.se.eecs.uni-kassel.de/se/

1 Motivation

In usual software engineering approaches, object oriented modeling with the UML is meanwhile state of the art. In this area, object orientation has been extremely beneficial: object oriented modeling allows to structure the application data in a way that eases maintenance and that allows to distribute the responsibility for certain functionality among the participating objects or components. Finally, object oriented techniques enable the use of modern design pattern that further improve flexibility and maintainability.

In the area of embedded systems, languages like VHDL are used to describe hardware circuits, directly. C is used to program micro controllers. Programable logic controllers are programmed in some kind of assembler, using function block diagrams or Pascal like structured text. Object oriented concepts are widely considered as inefficient and unsafe. Object oriented concepts usually imply some kind of pointer concept and dynamic organization of heap memory. Pointers may be null or point to already freed memory cells. The heap consumption may grow uncontrollably.

Since space restrictions and safety issues are of major concern in the area of embedded systems, this area does not yet widely use object oriented concepts. This has the drawback that the tremendous benefits of object oriented concepts such as flexibility in modelling, improved maintenance, larger manageable system complexity and the usage of design pattern technology are not available for the area of embedded systems.

We believe that in order to mature beyond the technology of the 80s and to catch up to the state-of-the-art in software development, embedded systems have to adapt object oriented concepts, too.

To prepare the ground for object oriented concepts in the area of embedded systems, this position paper exemplifies the use of object oriented modeling for the control software of a simple carousel storage system. It showns the systematic derivation from the modeled

components to an implementation on the basis of a programmable logic controller using structured text as programming language.

2 Example

To illustrate our ideas this paper uses the same LEGO model of a carousel storage as [GSZ04]. However, while [GSZ04] uses a micro controller programmed in Java, here we control the storage system with a PLC device, cf. Figure 1.

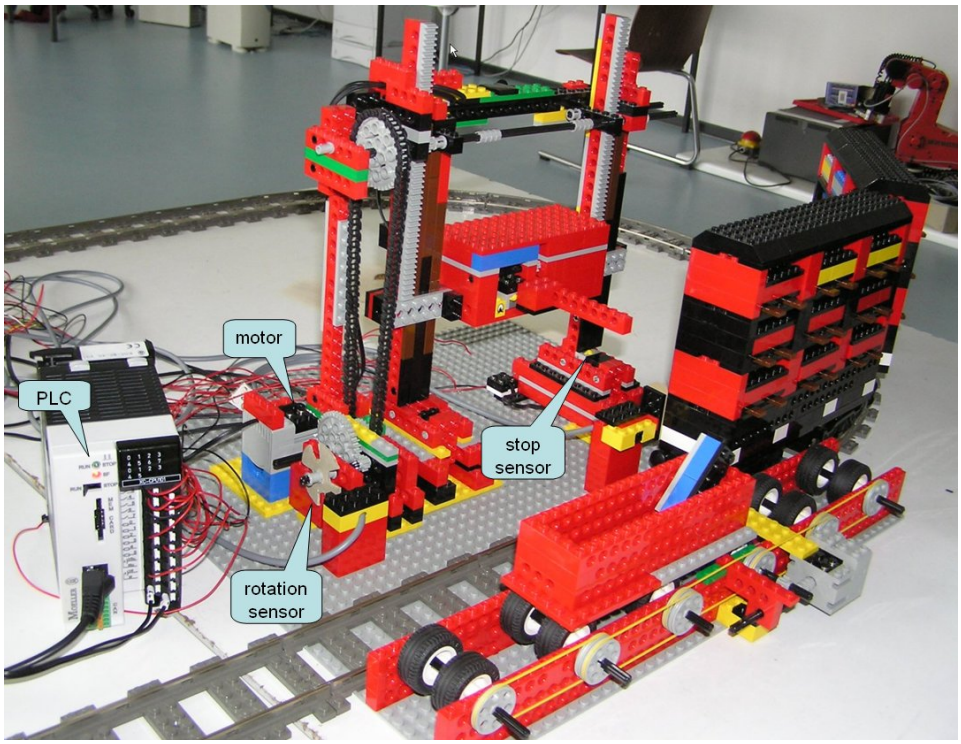

Figure 1: A simple LEGO model of a carousel storage

3 Object oriented model

In order to develop the control software for this LEGO model, we first developed an object oriented model for the components employed to control the storage, cf. Figure 2.

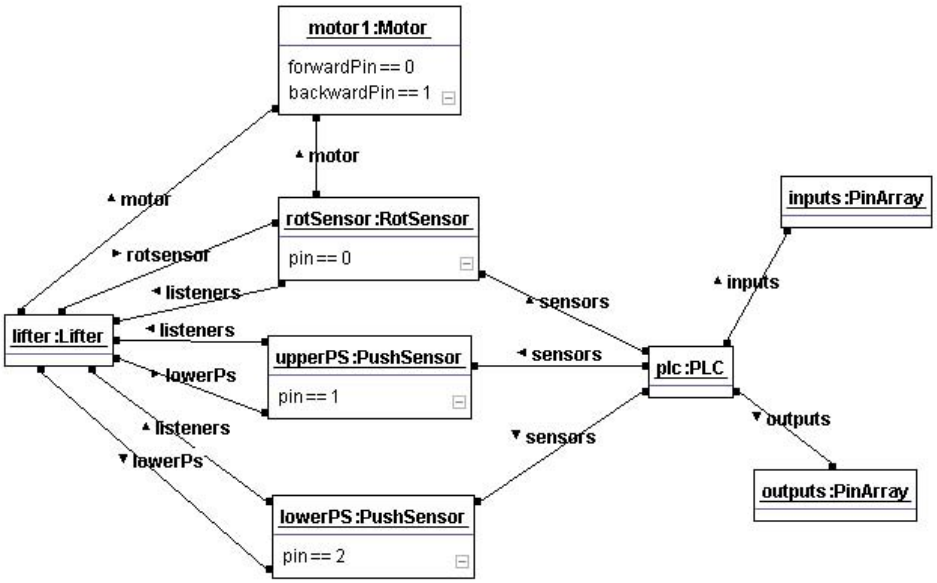

Figure 2: UML Object Diagram modeling some carousel components

We have modeled the lifter, its sensors, and its motor as distinct objects. In addition, the *plc* object represents the control of the PLC together with arrays of input and output pins. Each sensor knows the number of its corresponding input pin. Similarly, the motor object knows the pins letting the motor turn forward or backward.

Figure 3 shows the corresponding class diagram. Note, our design employs the well known observer design pattern at two places. First, the lifter subscribes at its sensors which inform the lifter when sensor values have changed. Second, the sensors subscribe at the PLC in order to be informed when new sensor pin values have been read.

Figure 4 shows a statechart modeling the main loop of the PLC. The PLC first initializes its components and then it sends a *evaluate* commands to all subscribed sensors. This step is iterated infinitely.

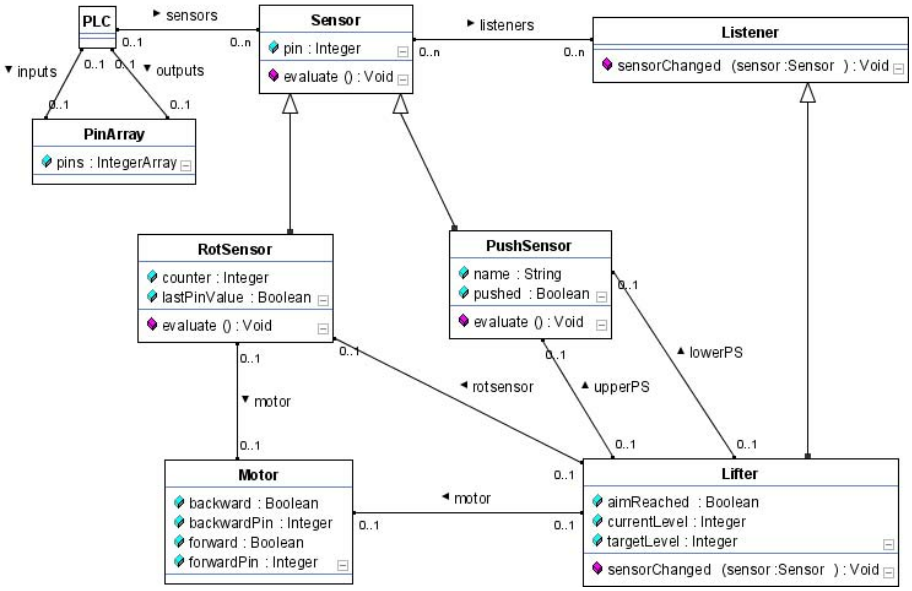

Figure 3: UML class diagram for the lifter components

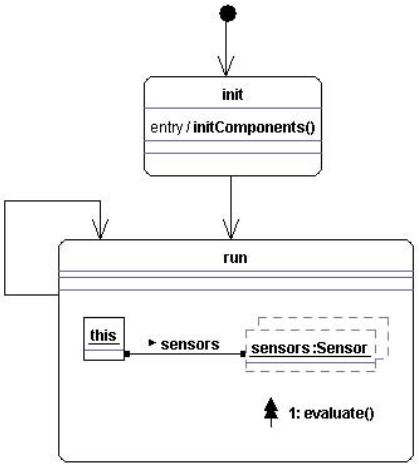

Figure 4: Statechart for the main PLC loop

As an example, for the *evaluate* method of a sensor, Figure 5 shows the method diagram modeling the behavior of a push sensor. On each call, the new sensor value is read from the input pins of the PLC. This is compared with the value from the previous call. If the value has changed, all listeners are informed.

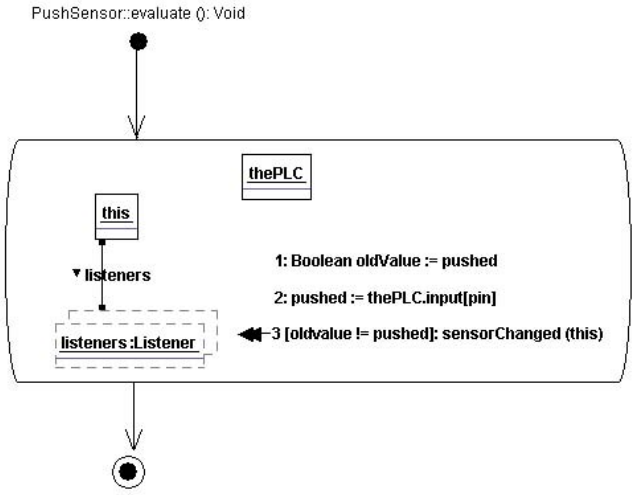

Figure 5: Method diagram for a push sensor

4 Implementation in PLC structured text

Our model employs various objects, links between objects, inheritance and method overriding, and the observer design pattern. This now has to be implemented in a PLC structured text, supporting a simple imperative paradigm only. In this section, we will outline how this may be done.

PLCs support different implementation languages. One of it is the so-called structured text which is close to a primitive Pascal dialect. Note, structured text is semantically equivalent to AWL (an assembler like dialect) and to function block diagrams (close to electronic circuit diagrams). We used a PLC programming environment that was able to show the same program in each of these notations.

The following rules state, how different object-oriented concepts can be translated into structured text:

Classes and Extensions: For each class we created a structured text *struct* declaring the fields of that class. In addition we declared a global array providing the storage for all instances of this class. We introduced a constant (labelled *maxNoOf* followed by the class name), limiting the maximal number of instances for that class. This constant is used to dimension the array of instances. Note, since our example employs only objects that represent physical components like sensors or motors and since we do not add such components at runtime, it is easy to provide these constants in this example.

Pointers: Pointers are just index numbers referring to a certain element of the array of instances of the type of the pointer.

To-many associations: If an object potentially has multiple neighbors, as e.g. a sensor may

have multiple listeners, we implement the corresponding *struct* field as an array of pointers, i.e. as an array of int. This again requires an array boundary. For simplicity reasons we use again the *maxNoOfXY* constant.

Thus, in structured text a PushSensor type looks like:

```
TYPE PushSensor :       # start of class declaration
STRUCT
    pushed : BOOL;      # attribute
    name : STRING;
    sensor : INT;       # pointer (to super class object)
END_STRUCT END_TYPE     # end of class declaration

VAR_GLOBAL CONSTANT
    maxNoOfPushSensors: INT := 4; # max no. of objects
    noOfPushSensors: INT := 0;    # no. of current objects
END_VAR

VAR_GLOBAL
    pushSensors: ARRAY [1..maxNoOfPushSensors]
       OF PushSensor;           # to-n association
    noOfPushSensors: INT := 0;  # no. of elements in assoc
END_VAR
```

Methods: Structured text provides functions with parameters. Unfortunately, methods must not be reentrant. This restriction has to be handled already in the OO model.

Obviously, methods are mapped on functions employing an additional first *this* parameter. To avoid name clashes, each method name is prefixed with its class name:

```
FUNCTION PushSensorIsPushed : BOOL VAR_INPUT
    this : INT;
END_VAR
IF 1 <= this AND this <= noOfPushSensors THEN
    PushSensorIsPushed := pushSensors[this].pushed;
ELSE
    PushSensorIsPushed := FALSE;
END_IF END_FUNCTION
```

Note, for safety reasons, our method implementations always check pointer validity before use. Actually, we just check whether the corresponding index number is in a correct range. This provides save use of pointers and thus it addresses one of the major objections against object oriented concepts.

Note, through sound encapsulation of field accesses, we are even able to provide bi-directional associations, where the pair of field update methods in the corresponding classes call each other, mutually.

Inheritance: Inheritance relationships may always be replaced by usual one-to-one associations. If this is done, the method implementations have to take care of problems like accessing super class attributes, etc. For example, the struct for the PushSensor class contains a field named sensor, cf. above. This is the reference to the object storing the super class fields of a given PushSensor. Accordingly, the struct for type sensor has field for each possible subclass. At runtime only one of these is employed:

```
TYPE Sensor : STRUCT
    pin : INT;
    pushSensor : INT := -1;  # pointer to subclass (null)
    rotSensor  : INT := -1;  #    "     "    "       "

    listeners : ARRAY [0..maxNoOfListeners] OF INT;
    myNoOfListeners : INT := 0;
END_STRUCT END_TYPE
```

Method overriding: Our example uses method overriding e.g. to implement the Observer design pattern. This means, the method evaluate is called on some sensor. This sensor may either be a rotation sensor or a push sensor. Both classes provide their own implementation for method evaluate. Depending on the runtime type, the correct implementation has to be invoked. This is achieved by a special dispatcher method implementation in the super class:

```
FUNCTION SensorEvaluate : INT VAR_INPUT
    this : INT;
END_VAR VAR
    subObj: INT;
END_VAR
subObj := SensorGetPushSensor (this);
IF  subObj >= 0 THEN
    PushSensorEvaluate(subObj);
ELSE
    subObj := SensorGetRotSensor (this);
    IF subObj >= 0 THEN
        RotSensorEvaluate(subObj);
    END_IF;
END_IF; END_FUNCTION
```

The sensor object is asked whether it has a PushSensor complement or a RotationSensor complement. Then, the appropriate method is called on the complement object.

Note, the techniques we use to implement the object oriented concepts in imperative structured text are pretty common compiler techniques used for the translation of usual object oriented languages.

5 Summary

This paper shows an example how object oriented models may be implemented even on a PLC device. This enables us to exploit the tremendous advantages of object oriented concepts even in the field of embedded systems. Safety concerns are addressed by limiting the maximal number of objects per type and by wrapping each use of a pointer within a null pointer check. These advantages are paid by additional memory and runtime consumption. However, we claim that the development process is shortened and that time-to-market and maintenance efforts are significantly improved. Finally one gains the flexibility provided e.g. by the use of design patterns.

The structured text example of this paper have been derived manually from our OO model, as a feasibility study. The code has been employed on a PLC controlling our LEGO carousel storage, successfully. The automatic generation of structured text from an OO model through our CASE tool Fujaba is current work at University of Paderborn, cf [EGSW04].

References

[DGMZ02] I. Diethelm, L. Geiger, T. Maier, A. Zündorf: Turning Collaboration Diagram Strips into Storycharts; Workshop on Scenarios and state machines: models, algorithms, and tools; ICSE 2002, Orlando, Florida, USA, 2002.

[DGZ02] I. Diethelm, L. Geiger, A. Zündorf: UML im Unterricht: Systematische objektorientierte Problemlösung mit Hilfe von Szenarien am Beispiel der Türme von Hanoi; in Forschungsbeiträge zur ”Didaktik der Informatik“ - Theorie, Praxis und Evaluation; GI-Lecture Notes, pp. 33-42 (2002)

[EGSW04] R. Eckes, J. Gausemeier, W. Schäfer, and R. Wagner: An Engineer’s Workstation to support Integrated Development of Flexible Production Control Systems; in Integration of Software Specification Techniques for Applications in Engineering (H. Ehrig, W. Damm, J. Desel, M. Große-Rhode, W. Reif, E. Schnieder, and E. Westkämper, eds.), vol. 3147 of Lecture Notes in Computer Science (LNCS), Springer Verlag, September 2004. (to appear).

[Fu02] Fujaba Homepage, Universität Paderborn, http://www.fujaba.de/.

[GSZ04] L. Geiger, J. Siedhof, A. Zündorf: μFUP: A Software Development Process for Embedded Systems; submitted to Dagstuhl Seminar MBEES, 2004

[KNNZ00] H. Köhler, U. Nickel, J. Niere, A. Zündorf: Integrating UML Diagrams for Production Control Systems; in Proc. of ICSE 2000 - The 22nd International Conference on Software Engineering, June 4-11th, Limerick, Ireland, acm press, pp. 241-251 (2000)

[Zü01] A. Zündorf: Rigorous Object Oriented Software Development, Habilitation Thesis, University of Paderborn, 2001.

AGGRESSIVE MODEL-DRIVEN DEVELOPMENT: SYNTHESIZING SYSTEMS FROM MODELS VIEWED AS CONSTRAINTS

Tiziana Margaria[1] and Bernhard Steffen[2]

[1]Service Engineering for Distributed Systems, Institute for Informatics,
University of Göttingen, Germany, margaria@cs.uni-goettingen.de
[2]Chair of Programming Systems, University of Dortmund, Germany
steffen@cs.uni-dortmund.de

Abstract: We propose an aggressive version of model-driven development (AMDD)[1], which moves most of the recurring problems of compatibility and consistency of software (mass-) construction and customization from the coding and integration level to the modelling level. AMDD requires a complex preparation of adequate settings, supporting the required automation. However, the effort to create these settings can be easily paid off by immense cost reductions in software mass-construction and maintenance. In fact, besides reducing the costs, AMDD will also lead towards a kind of normed software development, making software engineering a true engineering activity.

1 Motivation

1.1 The Problem

According to several roadmaps and predictions, future systems will be highly heterogeneous, they will be composed of special purpose code, perhaps written in different programming languages, integrate legacy components, glue code, and adapters combining different technologies, which may run distributed on different hardware platforms, on powerful servers or at (thin and ultra-thin) client sites. Already today's systems require an unacceptable effort for deployment, which is typically caused by incompatibilities, feature interactions, and the sometimes catastrophic behavior of component upgrades, which no longer behave as expected. This is particularly true for embedded systems, with the consequence that some components' lifetimes are 'artificially' prolonged far beyond a technological justification, since one fears problems once they are substituted or eliminated.

Responsible for this situation is mainly the level on which systems are technically composed: even though high level languages and even model driven development are used for component development, the system-level point of view is not yet adequately supported. In fact,

[1]We introduced the concept of Aggressive Model-Driven Design in our position statement at the Workshop on *Software Engineering for Embedded Systems: From Requirements to Implementation*, The Monterey Workshop Series, Chicago, Illinois, September 2003.

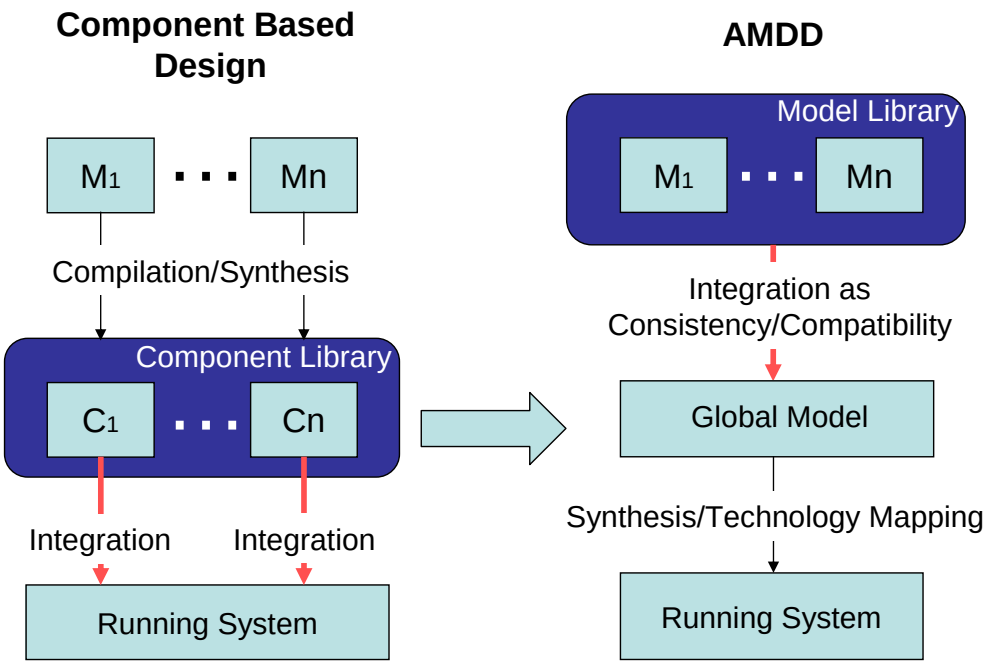

Figure 1: The AMDD Process

in particular the deployment of a heterogeneous systems is still a matter of assembly-level search for the reasons of incompatibility, which may be due to minimal version changes, slight hardware incompatibilities, or simply to hideous bugs, which come to surface only in a new, collaborative context of application. Integration testing and the quest for 'true' interoperability are indeed major cost factors and major risks in a system implementation and deployment.

Hardware development faces similar problems with even more dramatic consequences: hardware is far more difficult to patch, making failure of compatibility a real disaster. It is therefore the trend of the late '90s to move beyond VLSI to Systems-on-a-Chip (SoC) to guarantee larger integration in both senses: physically, compacting complex systems on a single chip instead of on a board, but in particular also projectually, i.e. integrating the components well before the silicon level, namely at the design level: rather than combining chips (the classical way), hardware engineers start to combine directly the component's designs and to directly produce (synthesize) system-level solutions, which are homogeneous at the silicon level. Interestingly, they solve the problem of compatibility by moving it to a higher level of abstraction.

1.2 AMDD: Aggressive Model-Driven Development

At the larger scale of (embedded) system development, moving the problem of compatibility to a higher level of abstraction means moving it to the modelling level (see Fig. 1): rather than using the models, as usual in today's Component Based Development paradigm, just as

a means of specification, which

- need to be compiled to become a 'real thing' (e.g., a component of a software library),
- must be updated (but typically are not), whenever the real thing changes
- typically only provide a local view of a portion or an aspect of a system,

models should be put into the center of the design activity, becoming *the* first class entities of the *global* system design process. In such an approach, as shown on the right side of Fig. 1,

- libraries should be established on the modelling level: building blocks should be (elementary) models rather than software components,
- systems should be specified by model combinations (composition, configuration, superposition, conjunction...), viewed as a set of constraints that the implementation needs to satisfy,
- global model combinations should be compiled (synthesized, e.g. by solving all the imposed constraints) into a homogeneous solution for a desired environment, which of course includes the realization of an adequate technology mapping,
- system changes (upgrades, customer-specific adaptations, new versions, etc.) should happen only (or at least primarily) at the modelling level, with a subsequent global recompilation (re-synthesis)
- optimizations should be kept distinct from design issues, in order to maintain the information on the structure and the design decisions independently of the considerations that lead to a particular optimized implementation.

With this *aggressive* style of *model-driven development* (AMDD), which strictly separates compatibility, migration, and optimization issues from model/functionality composition, it would be possible to overcome the problem of incompatibility between

- (global) models and (global) implementations, which is guaranteed and later-on maintained by (semi-) automatic compilation and synthesis, as well as between
- system components, paradigms, and hardware platforms: a dedicated compilation/synthesis of the considered *global* functionality for a specific platform architecture avoids the problems of incompatible design decisions for the individual components.

In essence, delaying the compilation/synthesis until all parameters are known (e.g. all compatibility constraints are available), may drastically simplify this task, as the individual parts can already be compiled/synthesized specifically for the current global context. In a good setup, this should not only simplify the integration issue (rather than having to be open for all eventualities, one can concentrate on precisely given circumstances), but also improve the efficiency of the compiled/synthesized implementations. In fact, AMDD has the potential to drastically reduce the long-term costs due to version incompatibility, system migration and upgrading, and lower risk factors like vendor and technology dependency. Thus it helps protecting the investment in the software infrastructure. We are therefore convinced that this

aggressive style of model-driven development will become the development style at least for mass customized software in the future. In particular we believe that AMDD, even though being drastically different from state of the art industrial embedded system design, which is very much driven by the underlying hardware architecture right from the beginning, will change accordingly: technology moves so fast, and the varieties are so manifold that the classical platform-focussed development will find its limits very soon.

1.3 The Scope of AMDD

Of course, AMDD will never replace genuine software development, as it assumes techniques to be able to solve problems (like synthesis or technology mapping) which are undecidable in general. On the other hand, more than 90% of the software development costs arise world-wide for a rather primitive software development level, during routine application programming or software update, where there are no technological or design challenges. There, the major problem faced is software quantity rather than achievement of very high quality, and automation should be largely possible. AMDD is intended to address (a significant part of) this 90% 'niche'.

What does this mean? AMDD aims at making things that inherently *are* simple as simple as they should be. In particular this means that AMDD is (at least in the beginning) characterized by abstractions, neglecting interesting, but at a certain level of development unnecessary, details, like e.g. distribution of computation, methods of communication, synchronization, real time. General software development practices can be replaced here by a model and pattern-based approach, adequately restricted to make AMDD effective. The challenge for AMDD therefore is initially to characterize and then model specific scenarios where its effectiveness can be guaranteed. Typically, these will be application-specific scenarios, at the beginning rather restrictive, which will then be generalized and standardized in order to extend the scope of applicability.

1.4 Making AMDD work

In order to reach a practicable and powerful environment for AMDD there is still a long way to go:

- adequate modelling patterns need to be designed,
- new analysis and verification techniques need to be developed,
- new compilation/synthesis techniques need to be devised,
- automatic deployment procedures need to be implemented,
- systems and middleware need to be elaborated to support automatic deployment, and,
- at the meta-level, we need a theory for the adequate specification of the settings which support this style of development.

It should be noted, however, that there is an enormous bulk of work one can build upon. Thus there is room also for quick wins and early success: AMDD is a paradigm of system design, and as such, it inherently leaves a high degree of freedom in the design of adequate settings, which, as described in Section 3, can be successfully used already today.

In the following we will focus on the following main ingredients:

1. a *heterogeneous landscape of models*, to be able to capture all the particularities necessary for the subsequent adequate product synthesis. This concerns the system specification itself, the platforms it runs on together with their communication topology, the required programming style, exceptions, real time aspects, etc.
2. a rich collection of *flexible formal methods and tools*, to deal with the heterogeneous models, their consistency, and their validation, compilation, and testing.
3. *automatic deployment and maintenance support* that are integrated in the whole process and are able to provide 'intelligent' feedback in case of late problems or errors.

2 What We Can Build Upon

2.1 Heterogeneous Landscape of Models

One of the major problems in software engineering is that software is multi-dimensional: it comprises a number of different (loosely related) dimensions, which typically need to be modelled in different styles in order to be treated adequately. Important for simplifying the software/application development is the reduction of the complexity of this multi-dimensional space, by placing it into some standard scenario. Such reductions are typically application-specific. Besides simplifying the application development they also provide a handle for the required automatic compilation and deployment procedures.

Typical among these dimensions, often also called **views**, are

- the *architectural view*, which expresses the static structure of the software (dependencies like nesting, inheritance, references). This should not be confused with the architectural view of the hardware platform, which may indeed be drastically different. - The charm of the OO-style was that it claimed to bridge this gap.
- the *process view*, which describes the dynamic behavior of the system. How does the system run under which circumstance (in the good case)
- the *exception view*, which addresses the system's behavior under malicious or even unforeseen circumstances
- the *timing view*, addressing real time aspects
- the various *thematic views* concerned with roles, specific requirements, ...

Of course, UML tries to address all these facets in a unifying way, but we all know that UML is currently rather a heterogeneous, expressive sample of languages, which lacks a clear notion of (conceptual) integration like consistency and the idea of global dynamic behavior. Such aspects are dealt with currently independently e.g. by means of concepts like *contracts* [1] (or more generally, and more complicatedly, via business-rules oriented programming like e.g. in [8]). The latter concepts are also not supported by systematic means for guaranteeing consistency. In contrast, AMDD views these heterogeneous specifications (consisting of essentially independent models) just as constraints which must be 'solved' during the compilation/synthesis phase ([20]).

Another recently very popular approach is Aspect Oriented Programming (AOP) [9, 2], which sounds convincing at first, but does not seem to scale for realistic systems. The programmer treats different aspects separately in the code, but has to understand precisely the weaving mechanism, which often is more complicated than programming all the system traditionally. In particular, the claimed modularity is only in the file structure but not on the conceptual side. In other words, in the good case one can write down the aspects separately, but understanding their mutual global impact requires a deep understanding of weaving, and, even worse, of the result of weaving, which very much reminds of an interleaving expansion of a highly distributed system.

2.2 Formal Methods and Tools

There are numerous formal methods and tools addressing validation, ranging from methods for correctness-by-construction/rule-based transformation, correctness calculi, model checkers, and constraint solvers to tools in practical use like PVS, Bandera, SLAM to name just a few. On the compiler side there are complex (optimizing) compiler suites, code generators, and controller synthesizers, and other methods to support technology mapping. A complete account of these methods would be far beyond the purpose of this paper. Here it is sufficient to note that there is already a high potential of technology waiting to be used.

2.3 Automatic Deployment and Maintenance Support

At the moment, this is the weakest point of the current practice: the deployment of complex systems on a heterogeneous, distributed platform is typically a nightmare, the required system-level testing is virtually unsupported, and the maintenance and upgrading very often turn out to be extremely time consuming and expensive, de facto responsible for the slogan "never change a running system".

Still, also in this area there is a lot of technology one can build upon: the development of Java and the JVM or the dotnet activities are well-accepted means to help getting models into operation, in particular, when heterogeneous hardware is concerned. Interoperability can be established using CORBA, RMI, RPC, Webservices, complex middleware etc, and there are tools for testing and version management. Unfortunately, using these tools requires a lot of expertise, time to detect undocumented anomalies and to develop patches, and this for every application to be deployed.

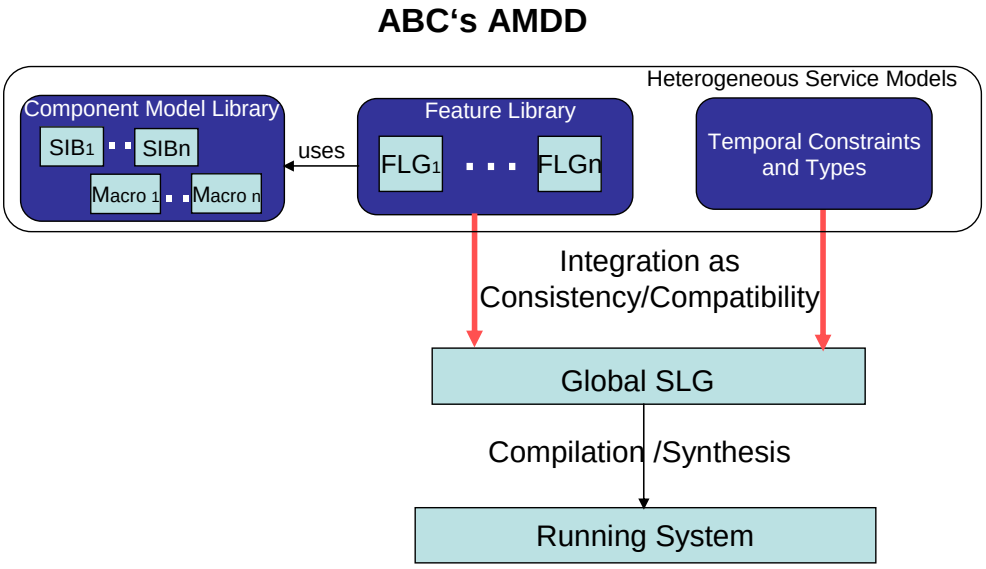

Figure 2: The AMDD Process in the ABC

3 A Simple AMDD-Setting

The Application Building Center (ABC) developed at METAFrame Technologies in cooperation with the University of Dortmund is intended to promote the AMDD-style of development in order to move the application development for certain classes of applications towards the application expert. Even though the ABC should only be regarded as a first step of AMDD development, it already comprises some important AMDD-essentials (Fig. 2.3):

1. *Heterogeneous landscape of models*: the central model structure of the ABC are hierarchical Service Logic Graphs (SLGs)[16, 15]. SLGs are flow chart-like graphs. They model the application behavior in terms of the intended process flows, based on coarse granular building blocks called SIBs (Service-Independent Building blocks) which are intended to be understood directly by the application experts [16] – independently of the structure of the underlying code, which, in our case, is typically written in Java/C/C++. The component models (SIBs or hierarchical subservices called Macros), the feature-based service models called Feature Logic Graphs (FLGs), and the Global SLGs modelling applications are all hierarchical SLGs.

   Additionally, the ABC supports model specification in terms of

   (a) two modal logics, to abstractly and loosely characterize valid behaviors (see [7]),

   (b) a classification scheme for building blocks and types, and

   (c) high level type specifications, used to specify compatibility between the building blocks of the SLGs.

   The granularity of the building blocks is essential here as it determines the level of abstraction of the whole reasoning: the verification tools directly consider the SLGs as formal models, the names of the (parameterized) building blocks as (parameterized) events, and the branching conditions as (atomic) propositions. Thus the ABC focusses

on the level of *component composition* rather then on component construction: its compatibility, its type correctness, and its behavioral correctness are under formal methods control [15].

2. *Formal methods and tools*: the ABC comprises a high-level type checker, two model checkers, a model synthesizer, a compiler for SLGs, an interpreter, and a view generator. The model synthesizer, the model checkers and the type checker take care of the consistency and compatibility conditions expressed by the four kinds of constraints/models mentioned above.

3. *Automatic deployment and maintenance support*: an automated deployment process, system-level testing [17], regression testing, version control, and online monitoring [4] support the phases following the first deployment.

   In particular the automatic deployment service needs some meta-modelling in advance. In fact, this has been realized using the ABC itself. Also the testing services and the online monitoring are themselves strong formal methods-based [18] and have been realized via the ABC.

In this sense, the ABC can be regarded as a simple and restrictive but working AMDD framework. In fact, in the ABC, composition/coordination of components as well as their maintenance and version control happen exclusively at the modelling level, and the compilation to running source code (mostly Java and C++) and deployment of the resulting applications are fully automatic.

4 AMDD Examples Based on the ABC

This section briefly sketches three systems in the light of our AMDD methodology: the *Online Conference Service*, the *Integrated Test Environment*, and *jETI*, an AMDD Platform for Remote Tool Integration that provides synthesis of heterogeneous tools on the basis of loose, abstract constraints.

4.1 OCS: The Online Conference Service

The OCS (Online Conference Service), see [11, 10, 13] for a description of the service and of its method of development, is a server-based Java application that customizes a strongly workflow-oriented application built with the ABC. It proactively helps authors, Program Committee chairs, Program Committee members, and reviewers to cooperate efficiently during their collaborative handling of the composition of a conference program. The service provides a timely, transparent, and secure handling of the papers and of the related tasks for submission, review, report and decision management. Several security and confidentiality precautions have been taken, in order to ensure proper handling of privacy and of intellectual property sensitive information. In particular, The service's capabilities are grouped in features [11] typically assigned to specific roles.

The OCS model structure, realized within the ABC, is characteristic for the AMDD approach. The *models* capture

- a user-oriented, top-down decomposition of services in terms of *features and roles*,
- a software-oriented, bottom-up composition of the service in terms of SIBs (Service Independent Building blocks), which are components reused across services of the same kind, and
- (de-)composition according to hierarchical considerations, via the reuse of sub-services (which build at a certain level sub-features) capsulated in so-called Macros. Moreover,
- the building blocks of all these (hierarchical) models are classified by taxonomies and
- the hierarchical feature- and SIB graphs are loosely constrained by temporal logic formulas.

4.2 ITE: The Integrated Test Environment

A different application of AMDD is the Integrated Testing Environment (ITE) [4, 14, 5] developed in a project with Siemens ICN in Witten (Germany). The ITE has been successfully applied along real-life examples of IP-based and telecommunication-based solutions: the test of a web-based application (the above mentioned Online Conference Service) and the test of an IP-based telephony scenario: Siemens' testing of the Deutsche Telekom's Personal Call Manager application, which supports among other features the role based, web-based reconfiguration of virtual switches. The core of the ITE is the *test coordinator*, an independent system that drives the generation, execution, evaluation and management of system-level tests. In general, it has access to all the involved subsystems and can manage the test execution through a coordination of different, heterogeneous test tools. These test tools, which locally monitor and steer the behavior of the software on the different clients/servers, are technically treated just as additional units under test.

In this project we practiced the AMDD approach at two levels:

- the modelling of the test environment itself, and
- the modelling of test cases.

The initial system-level test environment covered client-server third party applications interoperating with telecommunication switches and communicating over a LAN [17]. Soon, a new version of the ITE was required to be able to handle the next generation of applications, that from the engineering point of view had a completely different, and much more complex profile [14]. This meant a new quality of complexity along three dimensions:

- testing over the internet,
- testing virtual clusters, and
- testing a controlling system in a non-steady state (during reconfiguration).

These changes turned out to be rather straightforward due to the AMDD structure: an evolution step that Siemens considered to be 'close to impossible' became this way a matter of a few weeks.

4.3 jETI: An AMDD Platform for Remote Tool Integration

jETI, a redesign of the Electronic Tools Integration platform (ETI) [21, 22], combines Eclipse with Web Services functionality in order to provide (1) lightweight remote component (tool) integration, (2) distributed component (tool) libraries, (3) a graphical coordination environment, and (4) a distributed execution environment.

A more detailed account of the background and the new distributed way of tool integration for ETI can be found in [12]. Our current version of ETI, jETI, exploits recent Java technology to further simplify the remote tool integration and execution, and it naturally flexibilizes the original coordination level by seamlessly integrating the Eclipse development framework.

However, not only the tool composition is under model-driven control. All the tool functionalities are taxonomically characterized by means of *ontologies*, similar to the techniques adopted for the Semantic Web. ETI supports a global classification, but users may also introduce their private classification scheme, which helps them to quickly identify the tools relevant for certain applications. In fact, the requirement for this organizational support of tool functionalities was a result of a common project with the CMU, aiming at introducing a larger variety of formal verification tools in the undergraduate curricula.

5 Conclusions and Perspectives

We have proposed an aggressive version of model-driven development (AMDD), which moves most of the recurring problems of compatibility and consistency of software (mass) construction and customization from the coding and integration to the modelling level. Of course, AMDD requires a complex preparation of adequate settings, where the required compilation and synthesis techniques can be realized. Still, the effort to create these settings and their (application dependent) restrictions can be easily paid off by immense cost reductions in software mass construction and maintenance. In fact, besides reducing the costs, aggressive model-driven development will also lead (more or less automatically) to a kind of normed software development, making software engineering a true engineering activity.

This direction is also consistent with the perspective indicated by the joint GI-ITG position paper on *Organic Computing*[2] [19]: the blurring of borders between hardware and software (machines and programs) that initiated with embedded systems and with hardware/software codesign is going to reach a completely new dimension, where

- the systems are conceived, designed and implemented in terms of *services*,
- they are provided and used in a virtual space, and where

[2] GI, the Gesellschaft für Informatik and ITG, the Informationstechnikgesellschaft im VDE, the Verband der Elektrotechnik, Elektronik und Informationstechnik are the German counterparts of the ACM and IEEE, respectively.

- the distinction on where (local, global, at which node, on which hardware) and how (hardware, software, network, ...) the services are available is relatively inessential information.

In particular, according to availability or convenience, the provider of services can be changed and the provision of services is not a permanent contract anymore.[3]

Even though it is only a very first step, we consider the ABC a kind of proof of concept motivating the design of more elaborate aggressive model-based development techniques. The three examples in Section 4, which are quite different in nature, are very promising and indicate the potential of the AMDD approach.

References

[1] L.F. Andrade, J.L. Fiadeiro: *Architecture Based Evolution of Software Systems*, http://www.atxsoftware.com/ publications/SFM.pdf.

[2] AspectJ Website: http://eclipse.org/aspectj/

[3] CCITT/ITU-T: Recommendation 1201 - Principles of Intelligent Network Archtecture, 1992.

[4] 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), Grenoble (F), LNCS 2306, pp. 80-95.

[5] A. Hagerer, T. Margaria, O. Niese, B. Steffen, G. Brune, H.-D. Ide: *Efficient Regression Testing of CTI-Systems: Testing a Complex Call-Center Solution*, in *Annual Review of Communication*, Int. Engineering Consortium Chicago (USA), Vol. 55, pp.1033–1039, IEC, 2002.

[6] A. Hopper: The Royal Society Clifford Paterson Lecture: *Sentient Computing*, 1999.

[7] B. Jonsson, T. Margaria, G. Naeser, J. Nyström, and B. Steffen. Incremental requirement specification for evolving systems. Nordic Journal of Computing, vol. 8(1):65, Also in *Proc. of Feature Interactions in Telecommunications and Software Systems 2000*, 2001.

[8] JRules, ILOG. http://www.ilog.com/

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, J.Irwin: Aspect-Oriented Programming. Proc. of ECOOP, Springer-Verlag (1997).

[10] B. Lindner, T. Margaria, B. Steffen: *Ein personalisierter Internetdienst für wissenschaftliche Begutachtungs- prozesse* - In Proc. GI-VOI-BITKOM- OCG- TeleTrusT Konferenz on Elektronische Geschäftsprozesse (eBusi- ness Processes), Universität Klagenfurt, Sept. 2001, http://syssec.uni-klu.ac.at/EBP2001/ .

[11] T. Margaria: *Components, Features, and Agents in the ABC*, invited contribution to the volume *Components, Features, and Agents*, PostWorkshop Proceedings of the Dagstuhl Seminar on *Objects, Agents and Features* 7-21.3.2003, H.-D. Ehrich, J.-J. Meyer, and M. Ryan eds., appears in LNCS, Springer Verlag.

[12] T. Margaria: *Web Services-Based Tool-Integration in the ETI Platform*, Appears in SoSyM, Int. Journal on Software and System Modelling, Springer Verlag.

[3] This is a scenario that concretizes the idea of Sentient Computing [6].

[13] T. Margaria, M. Karusseit: *Community Usage of the Online Conference Service: an Experience Report from three CS Conferences*, 2nd IFIP Conference on "e- commerce, e-business, e-government" (I3E 2002), Lisboa (P), 7-9 Oct. 2002, in "Towards the Knowledge Society - eCommerce, eBusiness and eGovernment", Kluwer Academic Publishers, pp.497-511.

[14] T. Margaria, O. Niese, B. Steffen, A. Erochok: *System Level Testing of Virtual Switch (Re-)Configuration over IP*, Proc. IEEE European Test Workshop, Corfu (GR), May 2002, IEEE Society Press.

[15] T. Margaria, B. Steffen: *Lightweight Coarse-grained Coordination: A Scalable System-Level Approach*, to appear in STTT, Int. Journal on Software Tools for Technology Transfer, Springer-Verlag, 2003.

[16] T. Margaria, B. Steffen: *METAFrame in Practice: Design of Intelligent Network Services*, in "Correct System Design - Issues, Methods and Perspectives", LNCS 1710, Springer-Verlag, 1999, pp. 390-415.

[17] O. Niese, T. Margaria, A. Hagerer, M. Nagelmann, B. Steffen, G. Brune, and H. Ide. *An automated testing environment for CTI systems using concepts for specification and verification of workflows*. *Annual Review of Communication*, Int. Engineering Consortium Chicago (USA), Vol. 54, pp. 927-936, IEC, 2001.

[18] O. Niese, B. Steffen, T. Margaria, A. Hagerer, G. Brune, and H. Ide. Library-based design and consistency checks of system-level industrial test cases. In H. Hußmann, editor, *Proc. FASE 2001*, LNCS 2029, pages 233–248. Springer Verlag, 2001.

[19] *Organic Computing: Computer- und Systemarchitektur im Jahr 2010*, position paper of the VDE/ITG/GI. http://www.gi-ev.de/download/VDE-ITG-GI-Positionspapier

[20] B. Steffen. Unifying models. In R. Reischuk and M. Morvan, editors, *Proc. STACS'97*, LNCS 1200, pages 1–20. Springer Verlag, 1997.

[21] B. Steffen, T. Margaria, V. Braun: *The Electronic Tool Integration platform: concepts and design*, [22], pp. 9-30.

[22] *Special section on the Electronic Tool Integration Platform*, Int. Journal on *Software Tools for Technology Transfer*, Vol. 1, Springer Verlag, November 1997

Eine Integrierte Methodik für die Modell-basierte Entwicklung von Steuergeräte-Software

(work in progress)

Mirko Conrad, Heiko Dörr, Ines Fey

DaimlerChrysler AG,
Research E/E and Information Technology
Alt-Moabit 96a
10559 Berlin
{Mirko.Conrad|Heiko.Doerr|Ines.Fey}@DaimlerChrysler.com

Kerstin Buhr

TU Berlin,
Institute for Software Engineering and Theoretical Computer Science
buhr@cs.tu-berlin.de

Abstract: Die Herausforderungen bei der Entwicklung automotiver Steuerungssoftware lassen sich ohne eine geeignete Entwicklungsmethodik nicht mehr meistern. Einzelne Entwicklungsdisziplinen (z.B. Anforderungsermittlung, Modellierung, Test) werden zwar jeweils methodisch unterstützt, eine strukturierte Integration dieser spezialisierten Einzelmethoden ist derzeit noch nicht vorhanden. Der vorliegende Beitrag skizziert am Beispiel eines Modell-basierten Entwicklungsvorgehens Inhalt und Aufbau einer solchen integrierten Methodik. Deren Hauptbestandteile sind ein disziplinübergreifendes Informationsmodell, ein Baukasten aus aufeinander abgestimmten Vorgehensweisen für einzelne Entwicklungsaktivitäten sowie Hilfsmittel in Form von Checklisten, Templates etc.[1]

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

1 Motivation und Einleitung

Während die Komplexität und Leistungsfähigkeit eingebetteter Systeme im Kraftfahrzeug in der Vergangenheit stark durch die technischen Möglichkeiten (bspw. die Leistungsfähigkeit der Mikrocontroller) begrenzt wurde, wird aktuell der Entwicklungs- und Testprozess der Systeme und der darin eingebetteten Software zunehmend zum limitierenden Faktor [Sch98, Bec00]. Einem Teil dieser Limitierungen begegnet man seit Ende der 1990er Jahre durch den verstärkten Einsatz des *Modell-basierten Entwicklungsparadigmas*. Dieses vergleichsweise junge Entwicklungsvorgehen führt zu einer hohen Durchgängigkeit der verwendeten Modellierungsmittel und Werkzeuge. Ihre methodische Unterstützung befindet sich jedoch erst im Aufbau. Um die aktuellen Herausforderungen bei der Entwicklung automotiver Steuerungen meistern zu können, ist eine leistungsfähige methodische Unterstützung jedoch unverzichtbar.

Derzeit existieren vielfältige, spezialisierte Entwicklungsmethodiken für einzelne Entwicklungsdisziplinen, z.B. für Anforderungsmanagement, Modellierung/ Implementierung oder Test, die jede für sich dokumentiert sind. Das insgesamt vorhandene methodische Wissen ist damit in der Regel uneinheitlich dokumentiert.

Darüber hinaus, auch als Folge der uneinheitlichen Dokumentation, kann das Zusammenspiel der Einzelmethoden nur schwer beurteilt werden. Diese mangelnde Transparenz verdeckt Blindleistungen und methodische Inkonsistenzen. Das resultierende *Methoden-Patchwork* kann nur mit hohem Aufwand in Projekten zum Einsatz kommen. Die den jeweiligen Methoden innewohnenden Potentiale werden nur zu einem Teil wirksam.

Unintegrierte Teilmethodiken weisen i.d.R. Überlappungen auf und führen damit zu Mehraufwänden im Entwicklungsprozess. Unklare Bezüge und Inkonsistenzen an den Schnittstellen zwischen den Entwicklungsdisziplinen können darüber hinaus zu Qualitätsproblemen bzw. Informationslücken führen. Zudem werden die Kerninformationseinheiten der jeweiligen Entwicklungsdisziplin typischerweise detailliert und präzise beschrieben, während die Informationseinheiten aus den Überlappungsbereichen grobgranularer in Erscheinung treten. Inhaltlich gleiche Informationseinheiten werden durch die Teilmethoden nicht immer in konsistenter und einheitlicher Weise benannt.

Ziel einer *integrierten Methodik* ist die Unterstützung der methodischen Entwicklung von ECU Software über verschiedene Entwicklungsdisziplinen hinweg. Sie muß das Vorgehen bei der Entwicklung eingebetteter Software in der betrachteten Anwendungsdomäne beschreiben und gleichzeitig dem Systementwickler Hilfsmittel zur Anwendung der Methodik zur Verfügung stellen. Hierzu müssen die Einzelmethoden in geeigneter, einheitlicher Weise verfügbar gemacht und ihr Zusammenwirken definiert werden. Wegen der umfassenden Bedeutung einer integrierten Methodik muß diese so angelegt sein, daß sie über viele Projekte und über einen großen Zeitraum eingesetzt werden kann. Zentrale Anforderung an eine integrierte Methodik ist daher nicht nur deren Wartungsfreundlichkeit. Viel mehr muß sie eigenständige Mechanismen der Evolution beinhalten.

Eine integrierte Methodik muß dazu beitragen, Überlappungen und Inkonsistenzen zwischen den einzelnen Teilmethodiken zu identifizieren und darauf aufbauend zu beseitigen. Darüber hinaus muß der Sprachgebrauch in den Teilmethoden vereinheitlicht werden.

2 Aufbau einer integrierten Methodik

Das in der Praxis als erstes auftretende Hindernis für eine gegenseitige Abstimmung der Einzelmethoden ist die Vielfalt der Terminologien. Historisch gewachsene Begriffsbildungen fokussieren naturgemäß zunächst auf die jeweils eigenen Artefakte, mit der Folge, daß in Überlappungsbereichen nahezu immer konkurrierende Terminologien zum Einsatz kommen. Die Überwindung dieser Sprachbarrieren muß daher der erste Schritt auf dem Weg zu einer integrierten Methode sein. Grundlage der integrierten Methodik ist daher ein interdisziplinäres *Informationsmodell für die Modell-basierte Entwicklung*, welches alle wesentlichen in der Modell-basierten Entwicklung auftretenden *Informationseinheiten* und deren Zusammenhänge (*Beziehungen*) in abstrakter Form beschreibt. Dazu werden die jeweiligen Informationseinheiten, die im Rahmen der Anforderungsermittlung, der Modellierung und beim Test Verwendung finden, definiert und sowohl intradisziplinär (innerhalb einer Entwicklungsdisziplin) als auch interdisziplinär (zwischen unterschiedlichen Entwicklungsdisziplinen) in Bezug gesetzt.

Auf der Basis der abgestimmten Terminologie kann dann beschrieben werden, welche Rolle die jeweils bezeichneten Entitäten bei der Entwicklung von Steuergerätesoftware spielen. Dazu sind *Vorgehensweisen* für relevante Entwicklungsaktivitäten zu beschreiben. Eine einzelne Vorgehensweise ist dabei durch eine Folge von jeweils im Detail zu beschreibenden Arbeitsschritten definiert.

Zur Unterstützung der methodischen Durchführung der in den Vorgehensweisen aufgelisteten Arbeitsschritte müssen diese um *methodische Hilfsmittel*, wie z.B. Checklisten, Templates, Guidelines, Fragelisten, Formulare etc. ergänzt werden. Die Ausgestaltung der Hilfsmittel ist dabei von essentieller Bedeutung für die Akzeptanz der Vorgehensweise im einzelnen und auch der integrierten Methode im ganzen. Eine notwendige Erfolgsvoraussetzung ist etwa die direkte Einbindung der Hilfsmittel in die etablierten Werkzeuge.

Eine integrierte Methodik besteht damit aus folgenden Bestandteilen:

- (disziplinübergreifendes) Informationsmodell
- Vorgehensweisen für einzelne Entwicklungsaktivitäten
- methodische Hilfsmittel

Die einzelnen Bestandteile werden in den Folgeabschnitten ausführlicher beschrieben

2.1 Informationsmodell

Das Informationsmodell beschreibt die in den verschiedenen Entwicklungsdisziplinen und -phasen entstehenden Entwicklungsobjekte sowie deren Abhängigkeiten und Anordnungen (z.B. Hierarchien). Instanzen eines Informationsmodells liefern demzufolge eine statische Sicht auf den zu einem bestimmten Zeitpunkt existierenden Zustand einer Systementwicklung, d.h. auf die in einem Projekt vorhandenen Entwicklungsdaten.

Informationsmodelle für die Systementwicklung wurden in der Vergangenheit hauptsächlich im Kontext des Requirements Managements untersucht [JHN+99, WW02] und, auf diesen Anwendungskontext beschränkt, punktuell in der Praxis eingeführt. Darüber hinaus wurden Informationsmodelle verwendet, um den Austausch von Modellinformationen zwischen verschiedenen Modellierungs- und Simulationswerkzeugen zu erleichtern [SM01]. Mit STEP-AP 233 [STEP] liegt ein Standard-Informationsmodell vor, das jedoch die spezifischen Entitäten der Modell-basierten Entwicklung nicht berücksichtigt. In der Entwicklung von Business-Anwendungen ist der Einsatz von Informations- bzw. Referenzmodellen ebenfalls der erste Schritt hin zu einer IT- Unterstützung von Geschäftsprozessen.

Eine Herausforderung liegt in der sinnvollen Verlinkung der Informationseinheiten der unterschiedlichen Entwicklungsdisziplinen (z.B. Verlinkung von Anforderungen und Testbeschreibungen) und damit der engen Integration der Entwicklungsdisziplinen. Obwohl einige technische Lösungen dafür bereits verfügbar sind, fehlt eine methodische Herangehensweise, die den Prozess der Verknüpfungsfindung klärt. Unserer Erfahrung nach hat sich hier ein Ansatz bewährt, der die explizite Identifizierung einzelner Informationseinheiten der spezifischen Entwicklungsdisziplinen sowie deren Beziehung untereinander in den Mittelpunkt stellt. Wir nutzen für diese Herangehensweise ein disziplinübergreifendes Informationsmodell. Wir tragen dieser Tatsache durch die Verwendung von Informationsmodellen für die relevanten Artefakte und Dokumente aller Entwicklungsdisziplinen Rechnung.

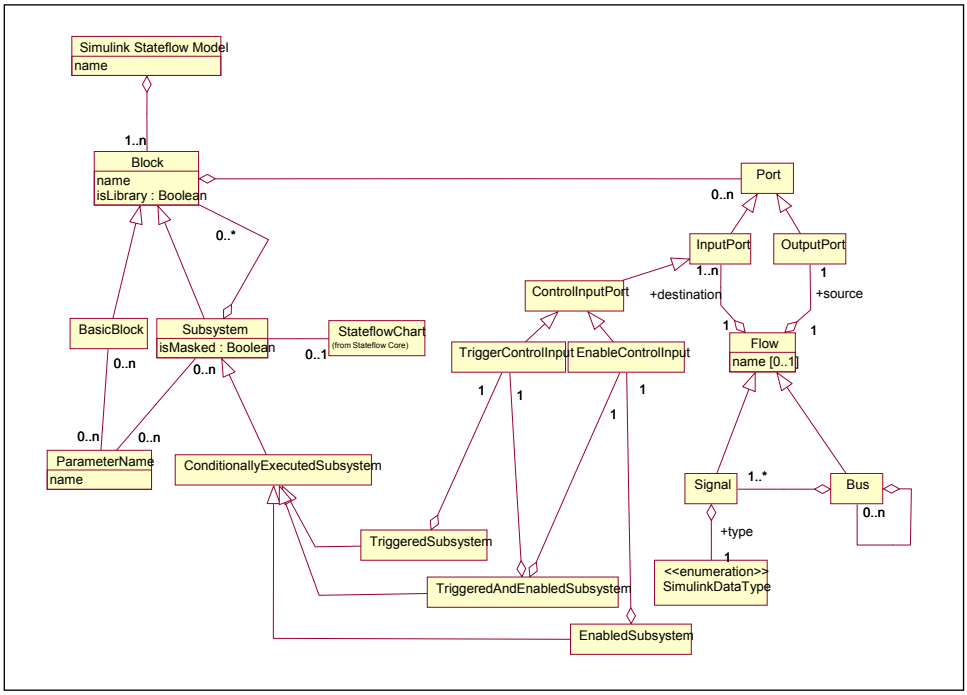

Abbildung 1: Informationsmodell Modellierung (Ausschnitt)

Zur Notation der Informationsmodelle werden UML-Klassendiagramme [UML20] bzw. davon abgeleitete Formalismen verwendet.

Informationsmodelle sind im Rahmen der integrierten Methodik 'Mittel zum Zweck'. Sie definieren und strukturieren den Diskursbereich. Darüber hinaus legen sie den Sprachschatz für die weiteren Bestandteile der integrierten Methodik fest.

Neben einer Fundierung der methodischen Integration verschiedener Entwicklungsartefakte kann ein derartiges Informationsmodell auch für die Evaluierung technischer Lösungen zur Abbildung des Entwicklungsprozesses dienen bis hin zu einer Bewertung von Werkzeuglösungen.

Im Rahmen der Modellierung werden beispielsweise Entwicklungsartefakte, wie Funktionsmodelle und deren Subsysteme bis hin zu Basisblöcken und Signalen, die in Subsystemen enthalten sind, betrachtet. Abbildung 1 zeigt einen entsprechenden Ausschnitt des Informationsmodells für die Entwicklungsdisziplin Modellierung.

2.2 Vorgehensweisen

Die Durchführung einzelner Entwicklungsaktivitäten soll durch *Schrittfolgen* und *beschreibende Texte* für die einzelnen Schritte erfolgen. Auf diese Weise kann das Vorgehen bei der Entwicklung eingebetteter Software in der betrachteten Anwendungsdomäne nachvollziehbar durchgeführt und projektübergreifend abgestimmt werden.

Eine Gesamtmethodik für die Modell-basierte Entwicklung ist ein dynamisches Gebilde. Zudem werden bei ihrer Anwendung projektspezifische Ausprägungen entstehen. Um die Handhabbarkeit zu gewährleisten, wir daher kein Vollständigkeitsanspruch (im Sinne einer monolithischen Gesamtmethodik) erhoben. Die Integrierte Methodik der Modell-basierten Entwicklung ist vielmehr ein *Baukasten* (construction kit) *aus verschiedenen Teilmethodiken*. Essentiell ist jedoch, dass die verschiedenen Methodikbausteine in einheitlicher und konsistenter Weise beschrieben werden. Die Konsistenz wird durch die Zugrundelegung des einheitlichen Informationsmodells unterstützt. Die textuellen Beschreibungen sind dabei auf die im Informationsmodell verwendeten Begrifflichkeiten abzustimmen. Auf diese Weise können verschiedene Methodikbausteine weitgehend unabhängig voneinander entwickelt und flexibel in den Methodikbaukasten integriert werden. Vorteil eines solchen Methodikbaukastens ist darüber hinaus, dass einzelne Teilmethodiken dann auch im Rahmen anderer Entwicklungsparadigmen (nicht Modell-basierte Entwicklung) eingesetzt werden können.

Für die Beschreibung der methodischen Vorgehensweise in den einzelnen Aktivitäten bestehen u.a. folgende Alternativen:

- Das in [Hei97] entwickelte Konzept der *Agenden* ermöglicht die explizite, feingranulare Repräsentation methodischen Wissens für eine begrenzte Anwendungsklasse. Agenden unterstützen die methodische Anwendung von Beschreibungstechniken durch detaillierte Schrittfolgen, sowie zusätzlicher Hilfsmittel in Form von Templates der verwendeten Notationen, die nur instantiiert werden müssen, sowie anwendungsunabhängige Validierungsbedingungen [GDH98]. Die Notation erfolgt in Form spezieller Tabellen.

- Das *Software Process Engineering Metamodel (SPEM)* [SPEM] kann zur Beschreibung eines konkreten SW-Entwicklungsprozesses oder einer Familie von SW-Entwicklungsprozessen auf Basis der UML-Notation benutzt werden. Es können u.a. die folgenden Grundkonzepte verwendet werden: Tasks, Techniques, Roles, Products, Phases.

Abb. 2 zeigt die Verwendung von Agenden zur Beschreibung einer methodischen Vorgehensweise im Testbereich an einem Beispiel. Dargestellt ist die Erstellung schnittstellenbasierter Klassifikationsbäume im Rahmen der Testbeschreibung mit der Klassifikationsbaummethode. Auf die zugehörige detaillierte textuelle Beschreibung der einzelnen Schritte (vgl. [Con04]) wurde aus Platzgründen verzichtet.

Nr.	Schritt	Validierungsbedingung(en)
1.	Extraktion potentiell testrelevanter Einflussgrößen (potentielles Interface) aus der Eingangs- und Parameter-Schnittstelle des Testobjektes ↗ Template 1.1.1: 'Roh-Klassifikationsbaum'	
2.	Anpassung an den zu testenden Sachverhalt: Ergänzung / Ausblendung von Einflussgrößen (effektives Interface) ↗ Template 1.1.2: 'Überlagerung mehrerer Bestandteile bei verrauschten Signalen' ↗ Template 1.1.3: 'Ersatzgrößen bei Reglerkomponenten' ↗ Template 1.1.4: 'Beschreibung von erwartetem Verhalten'	
3.	(Re-)Strukturierung des Roh-Klassifikationsbaumes ↗ Template 1.1.1: 'Roh-Klass.Baum'	
4.	Festlegung der Wertebereiche und Partitionierung der Einflussgrößen ↗ Template 1.1.5: 'Datentypspezifische Standardklassifikationen' ↗ Template 1.1.6: 'Fehlertypspezifische Standardklassifikationen'	• Konsistenz der Wertebereiche in der Testbeschreibung mit den entsprechenden Angaben im data dictionary des Modellierungs-werkzeuges bzw. im Lastenheft / in der Spezifikation • Disjunktheit, Vollständigkeit und Redundanzfreiheit der Klassen.

Abbildung 2: Agenda für den Modell-basierten Test (Auszug)

2.3 Hilfsmittel

Um die Anwendung der Methodikbausteine zu erleichtern und Erfahrungswissen verfügbar zu machen, sind dem Anwender der Integrierten Methodik Hilfsmittel zu ihrer Anwendung zur Verfügung stellen. Zu diesen methodischen Hilfsmitteln können je nach Entwicklungsaktivität z.B.

- Checklisten,
- Templates/Guidelines,
- Beispiele,

- Fragelisten oder
- Formulare
- Toolmentoren

zählen.

In Form solcher Hilfsmittel können die methodischen Essenzen bereits durchgeführter Projekte kondensiert und die Einarbeitung neuer Mitarbeiter erleichtert werden.

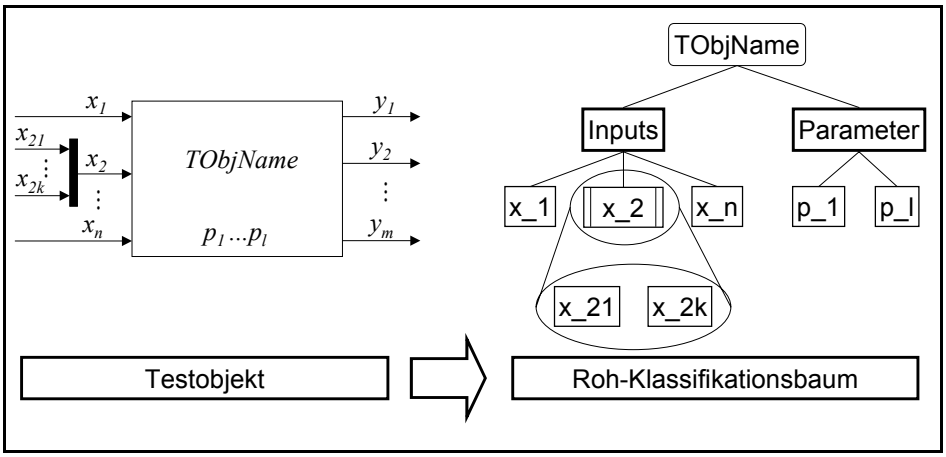

Abbildung 3: Template als Hilfsmittel für die Beschreibung von Testaktivitäten (Beispiel)

Die Beschreibung der Hilfsmittel erfolgt flexibel in Abhängigkeit von der Art des Hilfsmittels, typischerweise mit MS Office.

Im Rahmen des Agendenkonzeptes werden Hilfsmittel in Form Templates, d.h. von generischen Vorlagen für die einzelnen Schritte, verwendet. Im Rahmen von SPEM können einzelnen Informationseinheiten Guidances zugeordnet werden, die dem Anwender detaillierte Informationen über die entsprechende Informationseinheit geben. Guidances können kategorisiert werden.

Beispiel: Abbildung 3 zeigt das in in der Agenda in Abb. 2 referenzierte Template 1.1.1, das in generischer Form das Aussehen eines schnittstellenbasierten Klassifikationsbaumes für ein zu testendes Simulink-Subsystem beschreibt.

3 Umsetzungsstand

Teile des vorgestellten Konzeptes wurden im Rahmen von Forschungsaktivitäten bei DaimlerChrysler prototypisch umgesetzt und erprobt. Hierzu gehören u.a. die Erstellung des disziplinübergreifenden Informationsmodells, die Erstellung von Methodikbausteinen im Testbereich und Konzepte zur Verknüpfung von Anforderungen und Modellen.

Im Rahmen des Verbundvorhabens IMMOS [SSD+04] werden diese Arbeiten systematisch fortgeführt.

Literaturverzeichnis

[Bec00] P. Bechberger: Modellbasierte Softwareentwicklung für Steuergeräte. Automobiltechnische Zeitschrift / Motortechnische Zeitschrift, Sonderaus-gabe Automotive Electronics, Friedrich Vieweg & Sohn Verlagsgesellschaft mbH, Jan. 2000, S.16-24

[BD03] Kerstin Buhr, Heiko Dörr. Requirements Driven Quality Assurance. Proc. International Council on Systems Engineering (INCOSE'03), Washington (US), 2003.

[CFB04] M. Conrad, I. Fey, K. Buhr: Integration of Requirements into Model-based Development. Proc. International Workshop on Automotive Requirements Engineering (AuRE'04), Nagoya (J), Sept. 2004

[Con04] M. Conrad: Modell-basierter Test eingebetteter Software im Automobil - Auswahl und Beschreibung von Testszenarien. Dissertation, Deutscher Universitätsverlag, Wiesbaden (D), 2004

[GDH98] W. Grieskamp, M. Heisel, H. Dörr: Specifying Embedded Systems with Statecharts and Z - An Agenda for Cyclic Software Components. Proc of 1. Int. Conf. on Fundamental Approaches to Software Engineering (FASE '98), Lisbon (P), März 1998

[Hei97] M. Heisel: Imporving Software Quality with Formal Methods - Methodology and Machine Support. Habilitation, TU Berlin, Berlin (D), 1997

[JHN+99] G. John, M. Hoffmann, M. Nagel, C. Thomas, M. Weber: M.: Using a Common Information Model as a Methodological Basis for a Tool-supported Requirements Management Process. Proc. 9th Int. Symposium of the Int. Council on Systems Engineering (INCOSE'99), Brighton (UK), 1999.

[Sch98] H. Schmid: Entwicklung und Realisierung einer Teststrategie mit zugehöriger Testdatenbank für ein Kfz-Elektronik-Testsystem. Diplomarbeit, Stuttgart (D), März 1998

[SM01] E. Sax, K. D. Müller-Glaser: A seamless, model-based Design Flow for Embedded Systems in Automotive Applications. 1. Int. Symposium on Automotive Control, Shanghai (CN), 2001

[SPEM] OMG Software Process Engineering Metamodel (SPEM), 2000

[SSD+04] Schlingloff, H.; Sühl, C.; Dörr, H.; Conrad, M.; Stroop, J.; Sadeghipour, S.; Kühl, M.; Rammig, F. and Engels, G.: IMMOS - Eine integrierte Methodik zur modellbasierten Steuergeräteentwicklung. Proceedings, BMBF-Statusseminar „Software Engineering 2006", Berlin, 1.-3. Juli 200

[STEP] ISO/AWI 10303-233: Industrial automation systems and integration -- Part 233: Systems engineering data representation, 2003

[UML20] OMG Unified Modeling Language Specification, Version 2.0

[WW03] M. Weber, J. Weisbrod. Requirements Engineering in Automotive Development – Experiences and Challenges. In IEEE Software, vol. 20, no. 1, pp 16-24.

Eine offene Modellinfrastruktur für Architekturbeschreibung von Automotive Software Systemen
- Metamodellierung, Transformationen und Toolintegration -

Gabriel Vögler

DaimlerChrysler AG
Research and Technology
Software Service Center
Wilhelm-Runge-Str. 11, 89081 Ulm
gabriel.voegler@daimlerchrysler.com

Hajo Eichler

Fraunhofer Institut FOKUS
Model Driven Engineering
Kaiserin-Augusta-Allee 31
10589 Berlin, Germany
hajo.eichler@fokus.fraunhofer.de

Abstrakt: Durch die steigende Komplexität bei der Entwicklung von Automotive Software wird es zunehmend wichtiger, Entwicklungsinformationen über den gesamten Prozess hinweg zu formalisieren und systematisch zu integrieren sowie Teilaktivitäten zu automatisieren. Vor diesem Hintergrund wird der Ansatz eines Architekturinformationsmodells diskutiert, das als zentrale Datenbank wichtige Artefakte der unterschiedlichen Entwicklungsphasen speichert und integriert. Anhand einer prototypisch implementierten Modellinfrastruktur wird untersucht, inwiefern ein solcher Ansatz unter Verwendung von MDA-Technologien und domänenspezifischen Modellen für Automotive Software realisiert werden kann. Dabei soll das Speichern von Modellinstanzen, die halbautomatische Transformation von Modellen sowie die prozessweite Konsistenzprüfung und Traceability von Modellen unterstützt werden. Die Möglichkeiten der Metamodellierung und die konsequente Nutzung von Standards nach dem Vorbild der MDA soll dieses sog. Modell-Repository offen gegenüber den verwendeten Modellen und Werkzeugen machen. Für diesen Ansatz wird eine prototypische Implementierung mit Hilfe der Modellinfrastruktur "medini" auf Basis der EAST-ADL vorgestellt.

1 Einleitung

Die stetig steigende Anzahl softwarebasierter Funktionen im Fahrzeug, deren zunehmende Vernetzung sowie die Heterogenität des Entwicklungsprozesses führen zu einer Komplexität, die immer schwieriger zu beherrschen ist. Vor diesem Hintergrund wird es zunehmend wichtiger, die Entwicklungsartefakte der einzelnen Phasen problemspezifisch zu formalisieren und systematisch zu verknüpfen, um Abhängigkeiten zu analysieren, globale Konsistenz sicherzustellen und nachgelagerte Entwicklungstätigkeiten teilweise zu automatisieren. In der Automobilindustrie werden diese Punkte unter anderem mit der durchgängigen Modellbasierung des Entwicklungsprozesses adressiert [Kl04, MH03]. Bei Architekturbeschreibungssprachen wie z.B. der EAST-ADL [Lö04, Th03] werden den beteiligten Fachdisziplinen formale (i.d.R. grafische) Modelle zur Verfügung gestellt, die ihren Problembereich fokussieren und von anderen Realisierungsdetails abstrahieren. Auf diese Weise werden unterschiedliche Belange deutlich getrennt, was eine höhere Nebenläufigkeit von Entwicklungsaktivitäten und einen hohen Wiederverwendungsgrad von Entwicklungsartefakten ermöglicht. Der entscheidende

Beitrag in Bezug auf die Komplexitätsproblematik entsteht aber durch die systematische Verknüpfung der Modelle aus den einzelnen Entwicklungsphasen in eine integrierte Modellbasis. Im Folgenden wird ein Architekturinformationsmodells diskutiert, das viele dieser Prinzipien adressiert. Anhand einer prototypisch implementierten Modellinfrastruktur wird untersucht, inwiefern ein solcher Ansatz unter Verwendung von MDA-Technologien und domänenspezifischen Modellen für Automotive Software realisiert werden kann.

2 Ein integriertes Architekturinformationsmodell

Wir verwenden bei unserem Ansatz ein prozessweites Architekturinformationsmodell (Abbildung 1), das in der Form eines zentralen Modell-Repositories realisiert ist. Dieses speichert alle für die Komplexitätsbeherrschung wichtigen Entwicklungsinformationen, die über Verfeinerungs- und Mapping-Beziehungen miteinander verknüpft sind. Das Informationsmodell selber ist dabei über ein Metamodell definiert, das an die eigene Entwicklungsmethodik angepasst werden kann. So können sowohl standardisierte Modelle als auch selbst spezifizierte Modelle verwendet werden. Das Metamodell enthält Constraints, die vom Repository laufend überwacht werden. Außerdem ist es über Modelltransformationen möglich, Teilaktivitäten zu automatisieren.

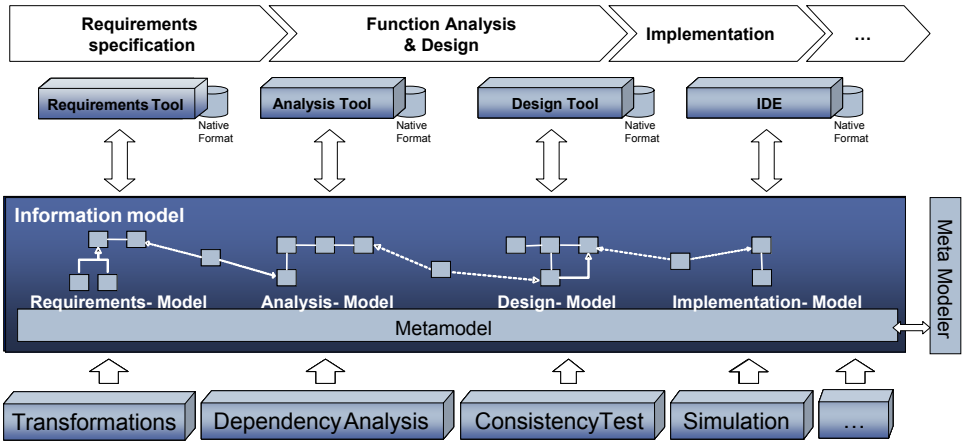

Abbildung 1: Das Architekturinformationsmodell

Das Metamodell definiert keine konkrete Notation für den Entwickler, sondern beschreibt lediglich die Zusammenhänge der einzelnen Artefakte und damit verbundene Regeln. Das Informationsmodell wird daher parallel zu der existierenden Toolkette aufgebaut. Die Notationen/Modelle der eingesetzten Entwicklungstools werden ebenso wie die nativen Speicherformate weiterverwendet. Die Tools werden lediglich um ein Plugin erweitert, das die relevanten Informationen (eine Untermenge der im Tool enthaltenen Informationen) entsprechend des Metamodells in dem Informationsmodell speichert. Über Modelltransformationen können diese Informationen halbautomatisch übersetzt bzw. verfeinert werden. Ein im Entwicklungsprozess nachgelagertes Tool kann das Er-

gebnis dann als Template verwenden (wird ebenfalls über ein Plugin geladen). Eine integrierte Modellinfrastruktur kann darüber hinaus weitere Mehrwertdienste anbieten, wie z.B. Abhängigkeitsanalyse, Konsistenzprüfung oder Simulation. Im Folgenden wird eine konkrete Infrastruktur vorgestellt, die diese Mechanismen auf der Basis von MDA-Technologien (insbesondere MOF [OM02] und UML 2 [BHK03]) und automobilspezifischen Metamodellen (EAST-ADL) realisiert.

3 MDA Building Blocks

Zunächst soll auf die Punkte eingegangen werden, die bei einer konkreten Ausgestaltung der MDA grundsätzlich zu klären sind. Dies umfasst die verwendeten (Meta-) Modelle, die eingesetzten Tools und die dabei verwendeten Standards sowie die Modelltransformationen.

3.1 Modelle: EAST-ADL

Eine wichtige Voraussetzung für ein integriertes Architekturinformationsmodell sind formale Modelle für die einzelnen Entwicklungsphasen, für die eindeutige Beziehungen definiert sind. Als ein mögliches Metamodell betrachten wir hierbei die EAST-ADL [Lö04, Th03]. Dabei handelt es sich um eine auf UML 2 aufsetzende Architekturbeschreibungssprache für Automotive, die im Rahmen des EAST-EEA-Projekts von den bedeutenden europäischen Automobilherstellern und Zulieferern entwickelt wurde.

Die EAST-ADL unterstützt alle Phasen eines Automotive-Softwarelebenszyklus mit verschiedenen Abstraktionsebenen (Abbildung 2). Dies umfasst z.B. im Rahmen der Anforderungsanalyse Modelle für die Erfassung von Funktionen aus Fahrersicht (inkl. Varianten). Für die funktionale Analyse gibt es Modelle, in denen die Struktur und das Verhalten dieser Funktionen implementierungsunabhängig modelliert werden können. In den nachfolgenden Modellen werden diese in Richtung einer Softwareimplementierung verfeinert (z.B. durch eine konkretere Definition von Funktions- und Signaltypen sowie Hierarchien und Signalflüssen) und mit technischen Informationen angereichert (z.B. bzgl. Zeitanforderung). In einem weiteren Schritt wird durch die Bestimmung von Varianten, der Auflösung von Hierarchien und der Zusammenfassung von Funktionen in Clustern (als Vorbereitung für die Verteilung auf Steuergeräte) ein konkretes System instanziiert. In einem letzten Schritt wird diese Architektur auf ein Plattformmodell (enthält die Steuergerätetopologie sowie die Schnittstellen und Dienste, die durch Betriebssystem, Middleware und die Hardware-Abstraktion angeboten werden) abgebildet, was zur letztendlichen Laufzeitarchitektur führt. Mit der EAST-ADL vergleichbare Architekturbeschreibungssprachen sind z.B. die SysML [Sy04] oder die AADL für den Luftfahrtbereich [SA03].

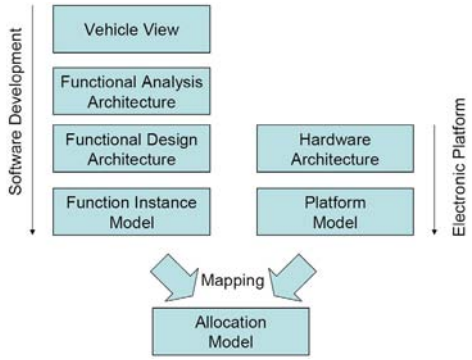

Abbildung 2: Abstraktionsebenen der EAST-ADL [Lö04]

3.2 Modellinfrastruktur: medini

Bei medini [Ok03] handelt es sich um eine Modellierungsinfrastruktur des Fraunhofer Institutes FOKUS und IKV++ Technologie AG, die auf der Model Driven Architecture (MDA) [MM03] der OMG basiert. So ist es mit dem Repository-Generator möglich, zu einem MOF-kompatiblen Metamodel ein Modellrepository zu generieren. Dazu werden die OMG Standards für die Schnittstellenbeschreibung (MOF2IDL, XMI, JMI) verwendet, die gleichzeitig durch eine C++ Implementierung realisiert werden. Diese automatisch generierten Modelldatenbanken zeichnen sich dadurch aus, dass sie eine Reihe von Mehrwertdiensten anbieten, wie z.B. Datenbankpersistenz und Versionsmanagement auf Elementebene. Ein weiteres Feature von medini ist die Unterstützung von OCL-Constraints, die im Metamodell mit eingewebt und im Repository überwacht werden.

Für die von der MDA angestrebten Modelltransformationen bietet medini mit dem MTG (MOF Transformator Generator) die Möglichkeit, das Skelett für einen Transformator ausgehend vom source-Metamodell aufzustellen, so dass nur die Implementierung der eigentlichen Abbildungsregeln (in C++) erstellt werden muss. Die Verbindungen zu den Repositories und das Auffinden der Elemente bleiben dabei für den Entwickler transparent.

3.3 Transformationen: Muster

Im Rahmen des BMBF-Projekts InPULSE wird bei DaimlerChrysler zurzeit ein Ansatz entwickelt, der sich peziell mit Transformationen zwischen Analyse- (ähnlich der FAA-Ebene) und Designmodellen (ähnlich der FDA-Ebene) für Kfz-Steuerungs- und Regelungssysteme beschäftigt. Grundlegende Idee hierbei ist es, den Zusammenhang zwischen diesen beiden Abstraktionsebenen auf der Basis von Mustern zu systematisieren. Zunächst werden dazu für die Analyseebene Muster aus der Regelungstechnik gesammelt. Diese beschreiben typische domänenspezifische Strukturen, wie z.B. einfache Actuator-Sensor-Beziehungen oder die klassischen Regelkreise. Für diese *Analysemuster* wird dann untersucht, wie sie durch Instanziierung von Designmustern auf der Designebene realisiert werden können und welche Variabilität dabei in Abhängigkeit von den nicht-funktionalen Anforderungen existiert. Auf dieser Grundlage werden Transformati-

onsregeln definiert, die für ein mit Analysemustern markiertes Analysemodell unter Angabe der nicht-funktionalen Anforderungen eine Designvorlage erzeugen. Darauf aufbauend wird ein Traceability-Modell entwickelt, mit dem Designentscheidungen anhand von verknüpften Musterinstanzen verfolgt werden können. Diese Arbeiten befinden sich noch in einer sehr frühen Phase und sind zum gegenwärtigen Zeitpunkt noch nicht in die hier vorgestellte Infrastruktur integriert. Die Transformationregeln, die in dem Beispiel dieser Arbeit verwendet werden, sind relativ einfach und durch einfache Programmierung auf der Basis des MTG umgesetzt.

4 Realisierung

Im Folgenden wird eine Realisierung des integrierten Architekturinformationsmodells auf Basis der EAST-ADL und der medini tool suite vorgestellt. Durch Verschmelzung von medini und der EAST-ADL wurde der Ansatz des oben beschriebenen Informationsmodells prototypisch implementiert. Exemplarisch wurde dabei ein Modellrepository für die drei EAST-Architekturen "Vehicle Feature Model" (VFM), "Functional Analysis Architecture" (FAA) und "Functional Design Architecture" (FDA) generiert, wobei sich die enthaltenen Informationen auf Strukturaspekte und Schnittstellen konzentrieren. Im ersten Fall ist eine Toolanbindung für das Requirement Tool Telelogic DOORS realisiert, in den beiden anderen Fällen für Rational Rose RealTime. Das EAST-Metamodell wurde darüber hinaus mit Constraints (OCL) angereichert, die vom Repository überwacht werden. Dabei handelt es sich zum einen um Constraints, die in textueller Form in der EAST-Spezifikation enthalten waren, zum anderen wurden aber auch eigene Constraints formuliert, die auf die Konsistenz zwischen den einzelnen EAST-Architekturen abzielen.

Für die Übergänge zwischen den Architekturen sind Transformationen auf der Basis des MTG implementiert. Die Transformationen sind dabei so realisiert, dass der Entwickler während der Transformation mit Hilfe von Dialogen manuell verschiedene Designentscheidungen treffen kann. Außerdem werden während der Transformation Traceability-Informationen gespeichert. Dazu wurde das Trace-Modell der EAST-ADL implementiert und an manchen Stellen erweitert. Dieses kann in einem XMI-fähigen UML Tool [OM03] visualisiert und dazu verwendet werden, entlang der Transformationspfade zu "navigieren".

5 Beispiel

Anhand von stark vereinfachten Modellen eines Adaptive Cruise Controllers (ACC) soll im Folgenden exemplarisch die Verwendung des Repositories für ein integriertes Architekturinformationsmodell demonstriert werden.
Abbildung 3 zeigt einen Überblick des Entwicklungsprozesses auf Basis unserer Infrastruktur und den angeschlossenen Tools (hier: Ausschnitt für Requirements und Analyse).

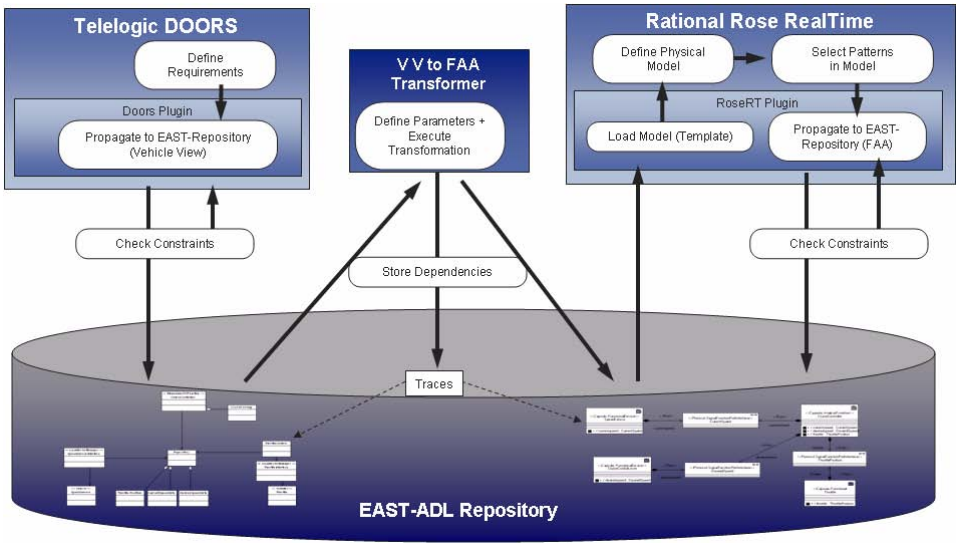

Abbildung 3: Entwicklungsprozess mit der Modell-Infrastruktur

Der Prozess beginnt mit der Requirements-Analyse, in der alle funktionalen und nicht-funktionalen Anforderungen sowie Varianten eines Fahrzeuges erfasst werden. Unsere Infrastruktur benutzt hierfür das in der Automobilindustrie weit verbreitete Tool Telelogic DOORS. Das DOORS-Requirementsmodell wurde dabei um einige Attribute erweitert, um alle Konzepte des EAST-Modells abbilden zu können. Über ein "Plugin" ist es somit möglich, die Requirements in das Informationsmodell zu propagieren (realisiert über ein DXL-Script). Abbildung 4 zeigt den ACC als *EFeature* entsprechend der EAST-Notation.

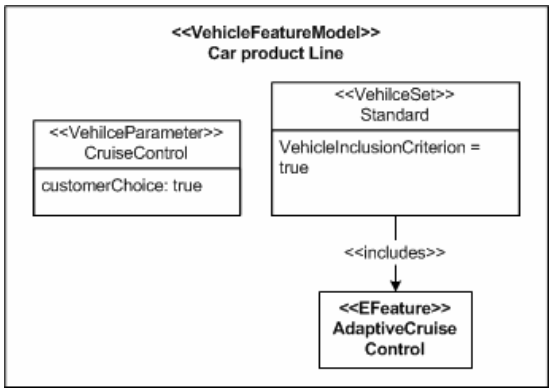

Abbildung 4: Vehicle Feature Model des Adaptive Cruise Controllers

Im nächsten Schritt wird ein abstraktes Funktionsdesign erstellt. Hierfür verwenden wir die Klassen- und Strukturgramme (in Verbindung mit Capsules) von Rational Rose

RealTime. Dazu wurde ein Plugin implementiert, das ein Modellieren der EAST-spezifischen Stereotypen der FAA erlaubt (vgl. Abbildung 5). Es werden dabei aus Entwicklersicht sowohl *AnalysisFunctions*, wie auch *FunctionalDevices* modelliert, die via Ports über Interfaces Daten austauschen, wobei diese Daten plattform-unabhängig als so genannte *DesignDataTypes* erstellt werden.

Das Funktionsdesign wird damit eingeleitet, dass in einem ersten Transformationsschritt aus den *EFeatures* - durch den Entwickler assistiert – ein Skelett für das Funktionsdesign erzeugt wird. Dabei ist vom Benutzer zu entscheiden, ob ein *EFeature* einem *FunctionalDevice* (also einem Sensor oder Actuator) oder eine *AnalysisFunction* (einem Steuerprogramm) entspricht. Dieses durch den Transformator erzeugte Template ist nach Abschluss der Abbildung im Repository verfügbar und wird durch das Plugin in Rational Rose RT eingelesen, um es dann zu verfeinern. In Abbildung 5 sieht man ein bereits verfeinertes FAA Model des Adaptive Cruise Controllers. Dialoge, Menüs und Toolbars unterstützen den Entwickler hierbei, um das Modell zu erstellen.

Um die Korrektheit des erweiterten FAA Modells zu testen bzw. um es dauerhaft abzuspeichern, wird es in das Repository propagiert, wobei alle Bedingungen des Metamodells eingehalten werden müssen. Dabei sind vor allem die Typkorrektheit, wie auch die Vielfachheiten von Assoziationen zu beachten.

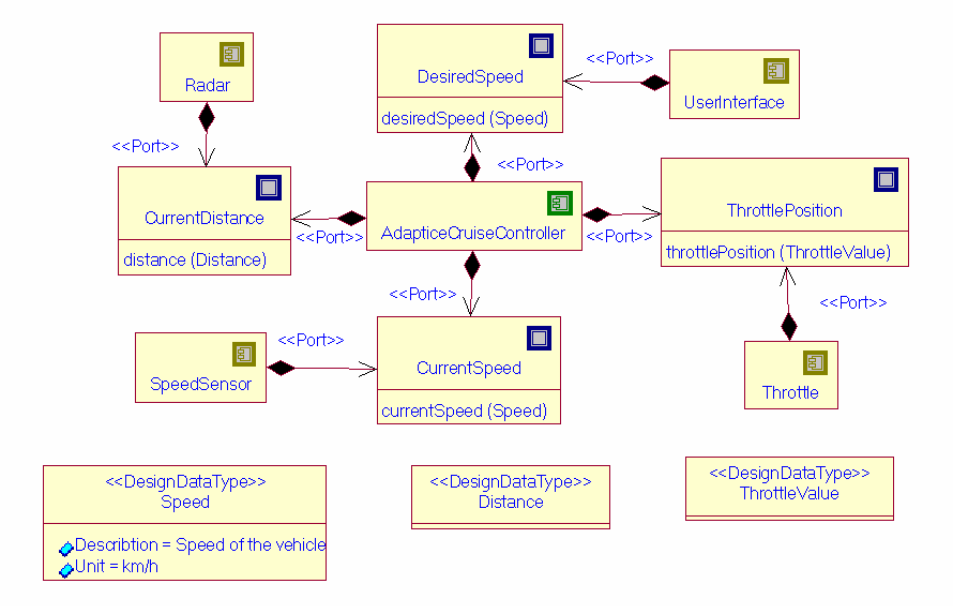

Abbildung 5: FAA Modell des Adaptive Cruise Controllers

Da das Repository auch die OCL 2.0 Constraints des Metamodelles überwacht, werden bei allen Schreibzugriffen ungültige Constraints dem Benutzer gemeldet.
Durch den folgenden Schritt wird das Analysemodell in ein Designmodell übertragen. Dabei werden ebenfalls die Klassen- und Strukturdiagramme von Rationale Rose Real-Time verwendet, wobei wieder über ein Plugin EAST-spezifische Stereotypen der FDA zur Verfügung gestellt werden.

Dieser Schritt beginnt mit dem Transformator FAA2FDA, um die Elemente der FDA in die Elemente der Designphase zu transformieren, wobei wieder ein Modellgerüst zur Weiterverarbeitung entsteht. Hierbei wird vom Entwickler erfragt, ob eine in FAA erstellte *AnalysisFunction* zu einer *CompositeSoftwareFunction* bzw. *ElementarySoftwareFunction* (abhängig von ihrer Granularität) transformiert werden soll. Des Weiteren werden *DesignDataTypes* auf *ImplementationDataTypes* abgebildet, wobei eine *TypeTransformation* angibt, wie die Verarbeitung erfolgt (vgl. Abbildung 6). Sollte eine solche *TypeTransformation* noch nicht existieren, so werden die erforderlichen Daten vom Entwickler in der Transformation abgefragt. Die entstehenden Implementierungsdaten werden mit dieser *TypeAssociation* verbunden, um die Typumwandlung später nachvollziehen zu können. Im Repository wird diese Information also als Trace festgehalten.

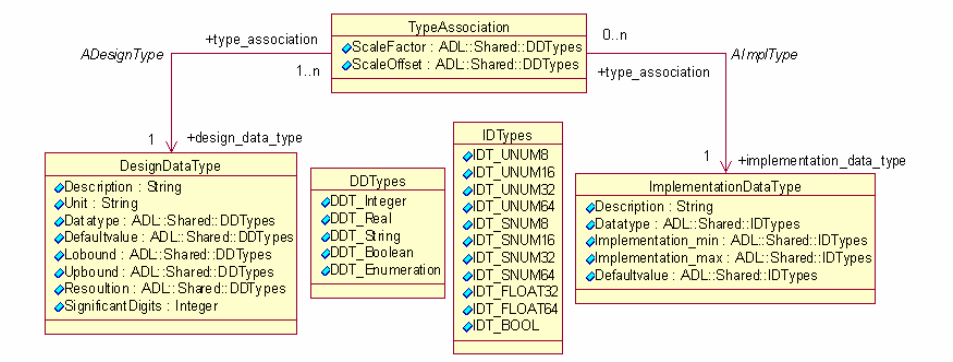

Abbildung 6: TypeAssociations

Nach diesem Transformationsschritt erfolgt das Laden des für die FDA im Repository bereitliegenden Templates in Rational Rose RealTime. Dies kann nun um die spezifischen Elemente der Designphase erweitert werden. Der Entwickler wird durch Menüs, Dialogfelder und Toolbars unterstützt. Abschließend kann das veränderte und erweiterte Modell wieder unter den schon beschriebenen Bedingungen (v.a. Modell- und OCL-Korrektheit bezüglich des Metamodells) in das Repository geschrieben werden.

Ein weiteres Feature ist eine XMI-Serialisierung aller im Repository abgelegten Trace-Informationen, um sie zu visualisieren. Hierbei wurde XMI gewählt um jedes beliebige XMI-fähige UML-Tool verwenden zu können und damit der Abhängigkeit von speziellen UML-Werkzeugen entgegenzuwirken.

Die Traces werden als getaggte UML-Abhängigkeiten (*realize* und *spezifiedBy*) dargestellt, die jeweils zwischen Klassen angelegt sind, die den Namen der verlinkten Elemen-

te besitzen. Um Namenskonflikte zu vermeiden, werden die Klassen ihrer Abstraktionsniveauherkunft entsprechend mit einem Stereotypen versehen (vgl. Abbildung 7). Auch werden die zuvor erwähnten *TypeAssociations* berücksichtigt und hierbei als UML-Klassen dargestellt, da sie einem mit Argumenten versehenem Trace entsprechen.

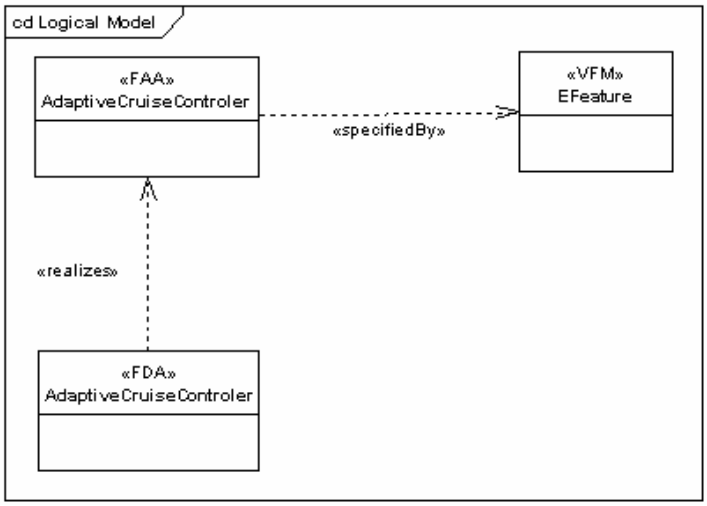

Abbildung 7 - Visualisierung von Traces im Informationsmodel (anhand von UML Klassen und Dependencies)

Zuletzt soll noch auf das oft auftretende Problem der Objektidentität bei der Integration von vorhandenen Tools in eine neue Infrastruktur eingegangen werden. Während ein Objekt – eine Instanz eines Metamodellelementes – im Repository eine eindeutige Identität hat – dies ist im MOF-Standard so definiert – müssen beim Anbinden von Modellierungswerkzeugen an eine solches MOF-basierte Modellrepository diese Identitäten im jeweiligen Tool mitverarbeitet werden. Im Beispiel tritt dieses Problem z.B. beim Laden des FAA-Modell-Templates in Rose auf, dass nach Verarbeitung wieder in das Repository geschrieben werden soll. Beim Schreibzugriff auf ein Repository muss geprüft werden, ob ein Modellelement des internen Rosebaumes nicht bereits im Repository existiert. Dafür werden die eindeutigen Objektidentitäten aus dem Repository mit in das Rose „mdl"-File eingebracht, so dass eine spätere Veränderung und Speicherung im Repository möglich ist.

6 Verwandte Arbeiten

Einige der bei uns behandelten Inhalte werden bei anderen Arbeiten unter dem Thema "Toolintegration" bzw. "Toolkopplung" im Zusammenhang mit der Entwicklung von eingebetteten Echtzeitsystemen behandelt. Der Schwerpunkt liegt herbei oft auf der technischen Integration der Entwicklungstools bzgl. Schnittstellen und Schnittstellentechnologien. So bietet z.B. *ToolNet* [ADS02] ein Framework für die Toolintegration auf der Basis vordefinierter Schnittstellen, die von den Entwicklungstools über Adapter

implementiert werden. Ein ähnlicher Ansatz wird bei EXITE verfolgt [Ex]. Bei uns liegt der Schwerpunkt in der Integration unterschiedlicher Entwicklungsartefakte auf der semantischen Ebene. Die Modell-Repositories bieten zwar über ihre Schnittstellen auch eine Grundlage für Toolintegration, die Kopplung ist aber relativ lose. Es ist z.B. nicht möglich aus einem Tool heraus Dienste eines anderen Tools aufzurufen und das Ergebnis direkt anzuzeigen. Außerdem enthält das Informationsmodell nur ausgewählte Informationen. Die Ansätze für direkte Toolkopplung können hier also ergänzend genutzt werden.

In [Br03] wird eine zu unserem Ansatz sehr ähnliche Idee skizziert. Es wird von der Integration der Tools DOORS, UML Suite und ASCET-SD auf der Basis eines gemeinsamen Metamodells mit dem Namen *Automotive Modeling Language* (AML) berichtet. Für die Transformation zwischen den Toolmodellen wird eine Transformationssprache namens *Bidirectional Object-oriented Transformation Language* (BOTL) verwendet. Der Beitrag ist sehr überblicksartig und geht wenig auf technische Details ein. Für einen genaueren Vergleich mit unserem Ansatz müssen wir noch weitere Publikationen sichten, die aus dieser Arbeit hervorgegangen sind. So stellt sich z.B. die Frage, ob für die Metamodellierung eine standardisierte Sprache wie MOF verwendet wird und ob das Metamodell fix ist oder an eigene Bedürfnisse angepasst werden kann. Die grundlegenden Konzepte scheinen aber ähnlich.

7 Zusammenfassung und Ausblick

Mit der hier vorgestellten Arbeit soll untersucht werden, inwiefern MDA-Technologien in Verbindung mit einem domänenspezifischen Architekturinformationsmodell einen Beitrag zur Lösung der Komplexitätsproblematik bei der Entwicklung von Automotive Software leisten können. Dazu wird eine prototypisch implementierte MDA-Infrastruktur auf der Basis der EAST-ADL mit exemplarischen Anbindungen der Tools DOORS und Rose RealTime evaluiert.

Es hat sich gezeigt, dass mit der EAST-ADL als Metamodell und MDA als Grundlage der Infrastruktur hilfreiche Mechanismen für die Komplexitätsbeherrschung zur Verfügung stehen. Exemplarisch wurde aufgezeigt, wie auf der Basis von Metamodellierung, der Formulierung formaler Constraints und der Definition von Transformationen mehr Konsistenz und Traceability im Entwicklungsprozess erreicht werden kann. Durch die konsequente Nutzung von Standards ist die vorgestellte Infrastruktur prinzipiell offen gegenüber unterschiedlichen Modellen und Werkzeugen.

Der produktive Einsatz gestaltet sich allerdings in der Praxis noch schwierig, da die neuen Standards von den Tool-Herstellern noch nicht im ausreichenden Maße implementiert sind. Dies bedeutet zurzeit für viele Tools umfangreiche Plugin-Programmierung. Zudem ist bei manchen Schlüsselkonzepten der MDA, wie z.B. bei den Transformationsdefinitionen, die Standardisierung noch nicht abgeschlossen. Die Philosophie der vollautomatischen Modelltransformationen ist nach unserer Ansicht nicht praktikabel. Hier müssen viele Designentscheidung bei der Transformation vom

Entwickler getroffen werden können, so dass die Transformationen entsprechend parametrisierbar und anpassbar sein müssen.

Das Architekturinformationsmodell wird redundant zu den bisherigen Speicherformaten der Tools gepflegt und enthält nur Informationen, die für die Mehrwertdienste (Traceability, Transformationen etc.) notwendig sind. Dies hat in der Praxis Vorteile, da die bestehende Tool-Landschaft beibehalten werden kann und keine radikale Umstellung notwendig ist. So ist es möglich mit einem "kleinen" Informationsmodell zu beginnen und dieses sukzessive parallel zu den normalen Entwicklungsaktivitäten auszubauen. Beispielsweise könnte nur der Vehicle Feature Model und die Functional Analysis Architecture der EAST-ADL verwendet werden.

Es sei jedoch daraufhin gewiesen, dass wir die EAST-ADL nicht als verbindlichen Standard für alle Phasen des Entwicklungsprozesses verstehen. Wir haben für den Prototypen die EAST-ADL verwendet, weil diese unter der Zusammenarbeit wichtiger Automobilhersteller entstanden ist und somit als Referenzmodell verstanden werden kann, in dem typische Automotive-Anforderungen berücksichtigt sind. In der Praxis sollen nach unserem Ansatz auch eigene Metamodelle verwendet werden. So ist es z.B. denkbar, nur einzelne Architekturen der EAST-ADL zu verwenden oder die Metamodelle zu erweitern.

Unser prototypisch implementiertes Informationsmodell konzentriert sich bisher nur auf Schnittstellen und Strukturaspekte. Als nächster Schritt wäre eine Erweiterung um Verhaltensaspekte denkbar. Für rein dynamisches Verhalten würde dies allerdings die Erstellung von MOF-Metamodellen für z.B. Matlab/Simulink bedeuten. Eine vollständige Simulink-Integration erscheint uns zurzeit jedoch nicht praktikabel. Zudem stellt sich die Frage, ob diese Informationen wesentlich zur Erfüllung der Ziele des Informationsmodells beitragen. Denkbar sind aber Links im Metamodell, mit denen z.B. *AnalysisFunctions* auf Simulink-Dateien verweisen können, die das Verhalten dieser Funktion beschreiben. Ebenso wäre ein Simulink-Plugin denkbar, das die Struktur- und Schnittstelleninformationen aus dem Informationsmodell für die Generierung von Hüllen für Simulink-Funktionsblöcke nutzt. Sinnvoller als die Erfassung von dynamischen Verhalten erscheint uns dagegen das direkte Speichern von (globalen) Zustandinformationen für ereignisorientiertes Verhalten im Repository.

8 Literaturverzeichnis

[ADS02] Altheide, F., Dörr, H., Schürr, A.: Requirements to a Framework for sustainable Integration of System Development Tools. In: H. Stoewer and L. Garnier (Eds): Proc. of the 3rd European Systems Engineering Conference (EuSEC'02). Frankreich, 2002, pp. 53-57. URL: http://www.es.tu-darmstadt.de/english/research/publications /download/EuSEC-02-Final.pdf.

[BHK03] Marc Born, Eckhard Holz, Olaf Kath: Softwareentwicklung mit UML 2, Addison-Wesley, München, 2003.

[Br03] Braun, P.: Metamodel-based Integration of Tools. In: Proceeding of ESEC/FSE 2003, TIS 2003 Workshop on Tool Integration in System Development. 2003.

[Ex] EXXTESY AG: EXITE. http://extessy.be-efficient.de/deutsch/content/leistungen/software/excite

[Kl04] Torsten Klein, Mirko Conrad, Ines Fey, Matthias Grochtmann (DaimlerChrysler AG): Modellbasierte Entwicklung eingebetteter Fahrzeugsoftware bei DaimlerChrysler. In: Bernhard Rumpe, Wolfgang Hesse (Hrsg.): Modellierung 2004. GI-Edition – Lecture Notes in Informatics (LNI), P-45. Köllen Verlag, Bonn, 2004, 31-42.

[KW01] Anneke Kleppe und Jos Warmer: The Object Constraint Language, Addison-Wesley, 2001

[Lö04] Lönn, H., et al.: Definition of language for automotive embedded electronic architecture description. EAST-EEA - Embedded Electronic Architecture, 2004. http://www.east-eea.net.

[MH03] Mutz, M., Harms, M, Horstmann, M. et al.: Ein durchgehender modellbasierter Entwicklungsprozess für elektronische Systeme im Automobil. Elektronik im Kraftfahrzeug. Baden-Baden, 2003.

[MM03] Joaquin Miller, Jishnu Mukerji: Model Driven Guide Version 1.0.1, OMG, 2003. http://www.omg.org/docs/omg/03-06-01.pdf.

[Mu04] Mutz, M., et al.: Ein durchgeheder modellbasierter Entwicklungsprozess für elektronische Systeme im Automobil, VDI Berichte 1789, 2003

[Ok03] Olaf Kath: medini - Werkzeuge für die modellgetriebene Softwareentwicklung, http://www.ikv.de/pdf/mediniWhitePaper.pdf, 2003

[OM02] Object Managment Group: Meta Object Facility, Version 1.4 (formal/2002-04-03) http://www.omg.org/technology/documents/formal/mof.htm

[OM03] OMG: XML Metadata Interchange (XMI) Specification Version 2.0. http://www.omg.org/cgi-bin/doc?formal/2003-05-02

[Sa03] The Society of Automotive Engineers: Avionics Architecture Description Language, 2003

[Sy04] SysML-Partners: Systems Modeling Language (SysML) Specification Version 0.85R1. http://www.sysml.org/artifacts/spec/SysML-v0.85R1-PDF-041011.zip

[Th03] Thurner, T., et al.: The EAST-EEA project – a middleware based software architecture for networked electronic control units in vehicles. In: Electronic Systems for Vehicles (VDI Berichte 1789), p 545 ff. VDI-Verlag, Düsseldorf, 2003.

Improving the Quality of Embedded Systems through the Systematic Combination of Construction and Analysis Activities

Christian Bunse, Christian Denger

Component based Software Engineering (CBE)
Fraunhofer Institute for Experimental Software Engineering
Sauerwiesen 6
67661 Kaiserslautern
christian.bunse@iese.fraunhofer.de
christian.denger@iese.fraunhofer.de

Abstract: The reliable attainment of quality requirements is still a weakness in model-based development projects, especially in the embedded systems domain. A major reason for this situation is the isolated use of construction and subsequent analysis activities. This paper describes a practical strategy for addressing this problem. The strategy relies on an effective combination of development and modelling guidelines as provided by the MARMOT method for component-based development of embedded and real-time systems and rigorous quality checks of the resulting models by means of the architecture-centric inspection (ACI) approach. This paper describes a strategy for combining analytic and constructive techniques in component-based development via an initial defect classification for UML design documents. The effects of such a combination are on the one hand high quality component implementations, and on the other hand a significantly increased confidence that the desired quality levels are attained.

1 Introduction

Component-based (CBSD) and object-oriented (OO) development of high quality systems has become a key issue for many industrial organizations especially in the embedded systems domain. Typically cited promises include higher reuse opportunities, increased development speed, improved software quality through lower failure rates, and lower costs associated with failure diagnosis and repair [DW98]. Because of these promises, OO and CBSD approaches have become the approach of choice for many development projects. Unfortunately, existing methods provide little guidance on how to achieve the promised benefits. In particular, quality requirements (i.e., non-functional requirements or NFRs) are often discarded until component testing or it is taken for granted that the method results, by definition, in high quality artefacts right from the beginning of a project [ABB01]. Both practices are detrimental. Quality, therefore, must be “built in” the components in a systematic manner right from the very beginning. In doing so, existing techniques, methods and tools have to be customized.

However, quality enhancing technologies are often limited to conventionally structured development methods. Prominent examples are techniques for early quality assurance, such as software inspections [Fag76]. Moreover, few techniques are actually tailored to the specificities of embedded systems. An exception is the MARMOT method, which offers a prescriptive and systematic approach for component-based product-line engineering with UML across the full software life-cycle. A major emphasis of the method is on quality because the very power of components makes it imperative that the reusable assets are of the highest possible quality. Clearly an organization that systematically produces or reuses poor-quality components will not fare well in the long run. For example, reusing a component does not only mean to reuse its functionality but also its flaws and quality issues. These might then affect the whole system.

While the construction activity in MARMOT uses UML-based modelling, the analysis activities involve the architecture-centric inspection (ACI) approach [TS+99]. ACI supports the systematic analysis of the developed modelling artefacts through a specific instantiation of a software inspection method. The major elements of the ACI approach consist of the optimal packaging of information and the systematic scrutiny of the packaged information using the perspective-based reading technique. This paper focuses on the integration & combination of UML based modelling (constructive) and ACI (analytic) via defect classification, and is structured as follows: Section two gives an introduction into systems modelling with UML. Section three introduces MARMOT and ACI as techniques for quality software development and section four outlines the followed integration strategy. Section five, introduces open research questions and provides a short summary and conclusion.

2 Modelling Embedded Components

In the following we describe the elements of the approach for modelling embedded components. Thus, principles, modelling diagrams for embedded components and methodological support are discussed.

2.1 Modelling Principles

Most existing CBSD methods only regard an entity as a component if it is implemented through a specific construct (e.g. a Java Bean), or modelled by using a particular abstraction (e.g. a component icon). In other words, being a component is regarded as an absolute property. In fact, being a component is a relative term rather than an absolute one. The term “component” indicates that one artefact (the component) may be a part of another artefact (another component), and certainly not that it is described in some particular abstraction. Composability may therefore be regarded as a key feature, and composition as a key activity in component-based development. Methods such as KobrA or MARMOT [Lai00, BC04] recognize this fact in that they advocate composition as the single most important engineering activity. A system can thus been viewed as a tree-shaped hierarchy of components, in which the parent/child relationship represents composition (i.e. a super-ordinate component is composed out of its contained sub-

ordinate components). Another, long established principle of software engineering is the separation of the description of what a software unit does (e.g., "specification", "interface", etc.) from the description of how it does it (e.g., "realization", "design", "implementation", etc.). This facilitates a "divide and conquer" approach to modelling in which a software unit can be developed independently. It also allows new versions of a unit to be interchanged with old versions provided that they do the same thing.

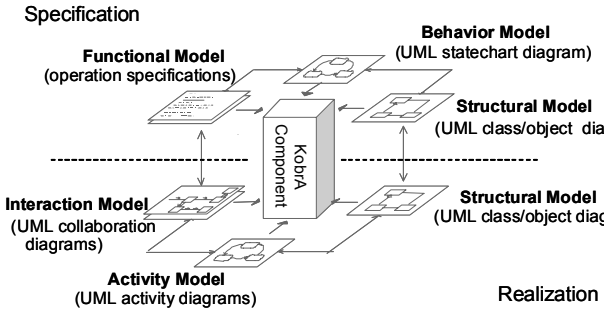

Figure 1: General Component Model

This principle is as important when modelling architectural components as it is when implementing them. A component modelled according to this principle is essentially described at two levels of detail - one representing a component's interface (what it does) and the other representing its body (i.e., how it fulfils the specified interface). Following this principle each component of a system can be described by a suite of UML diagrams as if it was an independent system in its own right (see Figure 1). This view on components is denoted by the term KobrA component or 'Komponent'. This separation allows developers who want to use an existing component or to replace one component with another to concentrate on the interface, neglecting the details of the body.

2.2 Embedded Components

The idea of modelling the components of a system using a standard suite of models is general applicable in that it can also be applied to non-software components. In detail this means, software and hardware components are treated in the same logical way. The concept of a component is extended by defining additional stereotypes: <<Electronics>>, <<Mechanics>>, <<Mechatronics>> in order to indicate the respective feasible device types and to provide the correct set of component specification artefacts (see Figure 2).

On the specification level a simplified view on all types of components, except the distinction between software and hardware components can be used (i.e., all components use the same suite of UML diagrams. However, at the realization level this view has to be specialized. At the realization level we have to distinguish between new in-house developments and component reuse (i.e., a decision has to be made if there is a product on the market that fulfils the component specification). In its simplest form, (re-)using an existing component is just a matter of instantiating it, and using its services in a way that conforms to their client-ship rules, defined through the specification.

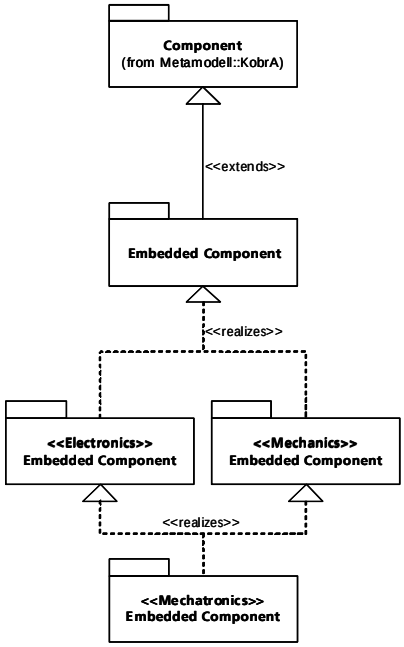

Figure 2: Electronic Component Model

Unfortunately, hardware-components may not always be realizable by COTS-components. Often, a specialized piece of hardware has to be developed in order to fulfil the desired specification. This is supported by using specific artefacts to describe the realization of hardware components.

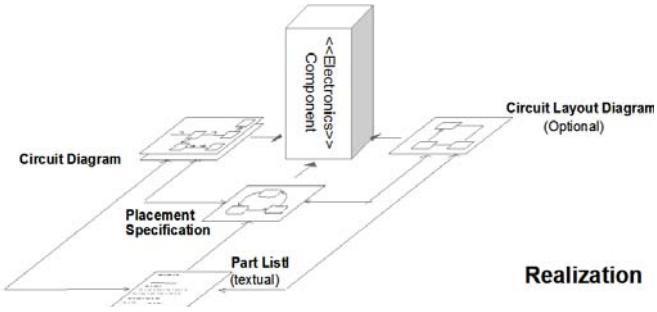

Figure 3: Realization of a <<Electronic>> component

A good example is the realization of an <<Electronics>> component, i.e., a component which realizes its interface by some kind of electronic circuit. The realization of such a component can be described using the following standard electrical engineering artefacts (see Figure 3):

1. The Circuit Diagram, which describes logical paths between the building blocks of a system through which an electrical current or signal is carried (see Figure 4).

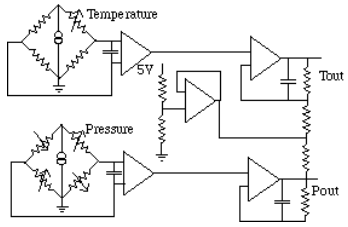

Figure 4: Circuit Diagram

2. The Part List, providing a textual description of needed building parts, stating the quantity, manufacturer and a unique identifier (e.g., 1x; Capacitator; HARDparts).
3. The optional Circuit Board Layout based on the decision if building parts are connected by wire or by a printed board layout.
4. The Placement Specification describes which part has to be placed to a specific location or how this part is connected to other parts.

The realization of a <<Mechanics>> component, a device represented by an assembly of mechanical parts (e.g., transmissions, gears, etc.) follows the same principle. In contrast to <<Electronics>> and <<Mechanics>> components, <<Mechatronics>> (e.g., limited slip couplings) components represent an assembly of mechanic and electronic parts. They aggregate the characteristics of <<Electronics>> and <<Mechanics>> components. In the realization, this dual nature is reflected by using the realization artefacts of both <<Electronics>> and <<Mechanics>> components. Optionally, artefacts describing the interaction between mechanic and electronic parts may be added.

2.3 Methodological Support

Based on the modelling of hardware and software components with UML, the MARMOT method (Method for Component-Based Real-Time Object-oriented Development and Testing) [BC04] provides methodological support for the development of embedded systems. In detail MARMOT:

- is completely based upon the KobrA method, and fully subsumes all of the method's principles and artefacts.
- treats software and hardware components in the same logical way. Hardware components are defined through KobrA specifications.
- is inherently aspect-oriented, it means that it supports a complete embedded system to be entirely considered from a particular perspective (e.g., NFRs).
- provides an iterative approach to system development and testing with real-time requirements, based on dynamic timing analysis, for non-critical real-time

systems, and a combination of static and dynamic timing analysis approaches for critical systems [Gro04].

- provides a top down approach to safety requirements through its in-built tracing facilities from user-level abstraction down to concrete designs [BC04].

3 Context

3.1 Development Process

A MARMOT project is based upon the following fundamental activities: (1) iteratively decompose the system into finer-grained parts that are individually controllable, this is termed "decomposition", and (2) reduce the level of abstraction to create representations of the system that come closer and closer to executable formats. In other words, every system is organized as a tree-shaped hierarchy of logical building blocks that have class-like and package-like properties. The class-like properties allow a component to have attributes, operations and behavioural features, whereas the package-like properties allow a component to represent a name space and act as container for a wide range of documents, concepts and other components.

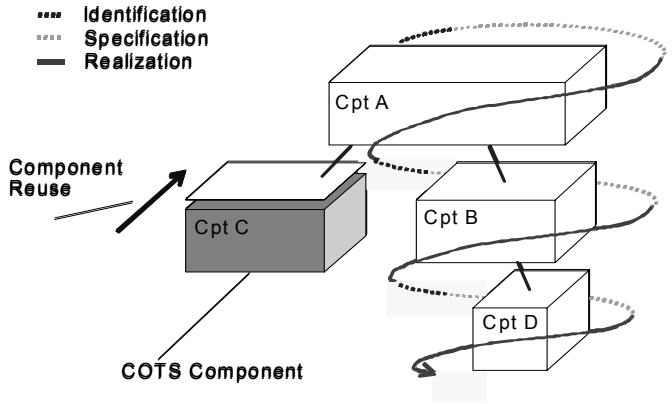

Figure 5: Development Process Overview

As demonstrated by the Cleanroom approach [MB+87], a recursive development process is inherently iterative. Figure 5 shows how the primary component engineering activities, when visualized in connection with the hierarchic product they generate can be regarded as leading to a spiral-based process. The final goal of the component reuse activities is to fully integrate a component that has been developed earlier outside the context of the tree (i.e., an external component).

To achieve this, the specification desired of the reusing component and the provided specification, offered by the pre-existing component, have to be brought into agreement. When such a situation exists, the reused component realizes, and usually also

implements the specification that is required by the reusing component, and the reused component is then fully integrated.

3.2 Architecture Centred Inspections

Concerning quality assurance we regard inspections as an integral part of development. The challenges in UML [OMG] based design in general and in MARMOT in particular is the huge amount of information that has to be inspected. Many various diagrams provide different views on the system and address different aspects. In contrast to other model based development approaches such as the Rational Unified Process [Kru98], the usage of the different diagram types is well defined in MARMOT. In that sense the purpose of every diagram and the stakeholders that are interested in the view on the system provided by the diagram are well defined.

However, the information is still too much to be inspected. In addition, Dunsmore et al. [TS+99] define delocalization of information as an additional challenge. To understand a function or a certain sub-system it is not sufficient to look at one place in the design but one has to consider many diagrams. Thus, the inspectors have to parse through the various diagram types in order to investigate a certain aspect of the system (e.g. the correctness of a component's interface). In contrast, if an inspection would focus on a certain diagram type, there is a high risk that essential information (e.g. with respect to the components interface) is described in another diagram and the inspector might miss too scrutinize these parts. In consequence, it is possible that crucial defects slip through the inspection. In order to mitigate this effect, and thus to improve the efficiency of the inspection, it is important to provide all relevant information to the author that is necessary to inspect a certain aspect.

The basic idea of the architecture-centred inspection approach (ACI) [BL02] is exactly to overcome the issue of delocalized information and to tackle the huge amount of information. The key idea behind ACI is that the inspectors are not focused on different diagram types but on slices of the design. Such a slice contains all relevant information with respect to a certain logical element of the system. A logical element is, for example, a component, a sub-system, a specific functionality, or a non-functional aspect. The slicing is done based on architectural relevant logical artefacts that comprise the core of the system. These elements are usually described in an architectural description. Therein, those elements of the system (hardware and software) that are most relevant are defined and their collaboration is described. The inspections are then built around those artefacts. If the inspector wants to verify a certain component defined in the architecture, the slicing approach tells him or her on which diagram types to look at, as the logical artefact might be described in many different diagrams of the design (e.g. the class diagram defines the general structure, the behavioural diagrams define the dynamics and the diagrams for embedded components the mapping to the hardware). This slice of the system provides the inspectors all relevant information to understand the logical artefact and resolves the problem of delocalized information. Moreover, it provides a clear focus for the inspectors so that they are not overwhelmed by the huge set of information.

In MARMOT a component represents a potential logical artefact of the system. Thus, the approach supports the inspection in a way that it is clearly defined which information is relevant to understand a component. For example, Figure 1 shows which types of diagrams an inspector has to inspect in order to verify a KobrA-component. In addition to the specification and realization models, the code, and the test cases implemented for the component need to be considered.

Diagrams of related components need to be considered as well, since these may contain essential information to understand the component under inspection. For this, the architecture is the main source of reference for the inspectors. In order to apply this concept to MARMOT it is important to consider the specific characteristics of embedded systems. That means that new diagram types have to be considered in the inspection approach (i.e. for electronics, mechanics and mechatronics), and that non-functional aspects of the design need more consideration (design specific aspects are for example: maintainability, testability reusability, and system wide non-functional aspects as performance, memory consumption, etc.). Providing inspectors with a well defined focus is one important success factor for inspections.

A second characteristic of ACI is that the inspectors are supported in finding defects within the chunk of information they have to scrutinize. Without explicit guidance on how to search for defects the inspection effectiveness will be restricted by the experience of the inspectors. Thus, an integral part of our approach is perspective-based reading (PBR) [BG+96, Lai00]. PBR comprises to main aspects: separation of concerns and active guidance. Separation of concerns means that the slice of the design is inspected from different perspectives. These perspectives represent stakeholders that are interested in the quality of a slice (e.g. a tester is interested in the testability of a component, a maintainer in its changeability and a developer might want to reuse the component and is therefore interested in its integratability and adaptability). Thus, each perspective addresses different quality aspects, which are tailored to the context of the MARMOT approach. This helps to reduce the overlap between different inspectors and further helps them to focus on specific issues, since the inspected slice of the design might still be too complex to address all quality aspects.

The second main element of PBR is active guidance by reading scenarios. For each perspective a reading scenario is defined. This scenario describes activities the inspector has to perform. While performing these tasks, inspectors have to answer questions with respect to the quality of the inspected information. The activities correspond to activities the related perspective would usually perform. For example, the tester scenario would provide guidance on deriving test cases from the inspected artefacts. A maintainer would try to incorporate some changes. Thus, the inspection produces not only a list of defects in the design but also additional documents that correspond to the performed activity (in the examples an initial set of system test cases, or a list of difficult to change parts). The additional output of the inspection can be reused in later phases as an initial starting point for the respective activities and thus, the time in the inspection for creating the artefacts is not wasted. The overall idea of actively working with the document during the inspection is that in that way the inspectors get a better understanding of the information they are looking at compared to just reading the information and then try to

answer related questions. Note that the degree of guidance given in the reading scenario depends on the experience of the reviewers. The less experienced the inspectors are in performing a certain task (create test cases) the more guidance in needed.

Applying PBR within MARMOT and in the context of ACI means that new perspectives have to be considered. In embedded system, non-functional requirements have to be addressed in more detail (e.g. a performance perspective or a memory consumption perspective could be created). In the context of reuse centred component based development it is also important to address related quality concerns by respective perspectives (e.g. a component user perspective that checks for interface correctness and integratability of the component in a new system context).

4 Structure for High Quality Components

To successfully combine constructive and analytic component development techniques it has to be analyzed which type of defects are addressed by the respective constructive and analytic development techniques. Based on this information synergies and overlaps of a combined approach can be identified and optimized. Thus, first it was analyzed which quality aspects a 'good' MARMOT model should have.

Based on this information and on studies concerning the addressed quality attributes of inspections [GG93, Lai00, TS+99, DR+02, BG+96, TS+99, Wie02, AP+02] we integrated ACI inspections, using PBR, into MARMOT to support the inspection of logical component entities, whereby the inspections focus on those quality attributes not explicitly addressed by MARMOTs modeling activities or tool support. However, a careful analysis and empirical evaluation of these quality attributes is an open issue, which has to be resolved in order to apply inspections in a highly efficient and effective manner. Starting with the analysis of the quality aspects of a good MARMOT model is the first step towards the definition of a MARMOT defect classification. Focusing on quality aspects, allows the combination of constructive and analytic approaches in order to provide 'optimized' techniques for a single quality aspect. The rationale for this is that both, the constructive and analytic techniques aim at ensuring these qualities either upfront or later by detecting defects that limit their fulfilment. An initial set of quality aspects of a "good" MARMOT model can be obtained from standards. These can serve as a baseline and can then be adapted by adding new aspects and tailoring the existing definitions to the context of the application domain. In the following we list an initial set of quality aspects to be used by MARMOT. The inverse definition of the quality attribute represents the defect classes. Note that we used the IEEE Std 730 standard as a starting point and tailored it to the MARMOT context.

Attribute	Definition
Consistency	There is no contradiction between the design elements on one level of abstraction (e.g. between class diagram and state charts)
Correctness	There is no contradiction between the design elements of different abstraction levels (e.g. realization and specification)
Completeness	All information specified on one level of abstraction is realized

	on the next lower level of abstraction.
Comprehensibility	The design of the components can be easily understood within a reasonable time frame
Testability	It is possible to derive reasonable test cases from the components' design, i.e. it is possible to validate that the components fulfills their specification
Maintainability	It is easy to change the design of the components
Feasabiltiy	The design of the components' can be implemented in code with the technologies at hand
Reusability	It is easy to reuse components in future projects (only those components that make sense to be reused).
Integratability	It is easy to integrate other components into the existing design.

Figure 6: Quality Attribute

When viewed in detail the constructive activities as well as the existing tool support address specific quality attributes concerning model consistency (between (1) UML models within specification and realization (partly), (2) specification and realization models, and (3) components), correctness (of refinements), visibility of components, and method adherence (e.g., completeness of models (diagram-wise), refinements, shaping of the component tree, etc.), Defects related to the quality attributes are already reduced at construction time. Thus, these quality attributes need less attention in the analytic activities.

The analytic activities focus on aspects that cannot be ensured by the constructive activities and its tool support. Thus, they focus on logical and most subtle defects. Testability, comprehensibility, reusability, feasibility, integratability, and maintainability are important issues in this regard, which can and should be checked early in development by adapted inspections. The PBR reading technique requires, according to the total quality management (TQM) paradigm, that every user or customer of an artefact has to check its quality. Concerning testability it has been proved beneficial to involve testers in checking the respective UML models [TS+99]. Similarly, inspectors should focus on the reusability of components as reuse is a major issue in component based development in general and MARMOT in particular. Finally, inspectors should address semantic issues in the design. Semantic defects cannot be detected automatically by tools and it is almost impossible to omit these defects in a constructive way. The detection of these defects is supported by the active guidance of the ACI approach. Of course, it should also be briefly checked in the inspection that the constructive guidelines were adhered during the creation process, as long as these are not ensured by a tool. However, MARMOT provides tool support that ensures method adherence; Thus, inspectors do not have to spend much effort concerning this issue and can concentrate on the identification of major defects which should be the objective of inspections.

5 Open Questions and Conclusion

By now we have integrated constructive and analytic activities within the MARMOT approach to systematically develop high quality components. However, there are still some open questions, which have to be resolved. With respect to the ACI inspections it

has to be investigated how the models related to hardware components can be inspected in an efficient way and which quality aspects have to be considered in the inspection of such components. Moreover, we need to discuss how inspections or related static analysis techniques can be used to address non-functional issues such as performance, memory consumption, maintainability, testability etc. early in the life-cycle. Here the main question is how to create suitable slices on the system that comprise all relevant information related to the non-functional aspects. Aspect orientation and architecture assessment techniques can provide initial hints on how to achieve this. Starting by identifying the qualities of 'good' MARMOT models was an initial step to come up with a first combination approach. However, a more detailed defect classification for UML models is needed in order to gain an even more efficient combination approach. Such a defect classification should also consider the potential defect types concerning the models of hardware components. Moreover, it is necessary to draw a link between defect types of the MARMOT models and the potential impact these defects might have. In other words, to focus the quality assurance activities, it is important to investigate how certain quality aspects (maintainability, testability, reusability, functional correctness, etc.) manifest themselves in the MARMOT design.

In the context of component based development it is also important to customize the inspection process to the characteristics of the environment. A crucial aspect of such tailored approaches is how a cost-effective enactment of quality assurance techniques might look like in the context of reuse. Thus, it is important to come up with a balancing model that tackles exactly the question on how to efficiently enact and perform software inspection to address the challenges and specialties imposed by the reuse intensive nature of component based development. Such a balancing model should help to answer the question which quality aspects should be inspected on reusable components and which should be addressed on a reused instance of the component [DT+04]. Finally, it has to be investigated how constructive and analytic techniques contribute to ensure certain quality aspects of a system as a whole (e.g., its reusability, maintainability, dependability, etc.) and of different development artefacts that describe the system (consistency, correctness etc. of the requirements, design and code artefacts).

With the rapid rate of innovation in component-based technology and paradigm switches in embedded system development, one might have expected to have seen significant improvements on how to ensure the quality of component-based or embedded software systems. In practice, however, this has rarely happened. There is still a lack of techniques to improve the quality of components. While many of the existing development methods focus on the early phases, this paper presents a combined strategy to support the construction and subsequent analysis of components, especially in the embedded system domain. The strategy is built upon existing technologies, but their combination allows to benefit from synergy effects and thus, represents an impetus for high quality component implementation, since prescriptive guidance can be given to developers on how to implement and analyze a component or parts of it. We currently plan an empirical study to close some of the open issues discussed in the previous section, and to investigate the return on investment of the suggested strategy. Moreover, we are looking for tools to support both approaches and automate steps as best as

possible. Both are necessary ingredients to drive the adoption of this approach in practical development situations.

References

[ABB01] Atkinson, C., Bayer, J., Bunse, C., et al: Component-Based Product Line Engineering with UML. Pearson, 2001.

[ABD+] Amiri, S, Bunse, C., Denger C., et al: MARMOT – Method for object-oriented and component-based embedded real-time system development and testing, http://www.marmot-project.org.

[AP+02] Aurum, A., Petersson, H., Wohlin, C.: State-of-the-Art: Software Inspections after 25 Years, Software Testing Verification and Reliability, Vol. 12(3), pp. 93-122, 2002

[BC04] Bunse, C., Choi, Y.: Behavioral Consistency Checks for Component-based Software Development using the KobrA Approach, Workshop: Consistency Problems in UML-based Development" at <<UML>> 2004.

[BG+96] Basili, V., Green, S., Laitenberger, O., Lanubile, F., Shull, F., Sorumgard, S., Zelkowitz, M.: The Empirical Investigation of Perspec-tive-based Reading, Empirical Software Engineering 1, pp. 133–164, 1996.

[BL02] Bunse, C., Laitenberger, O.: Improving Component Quality through the Systematic Combination of Construction and Analysis. In Proceedings of Software Quality Week Europe, Belgium, 2002.

[DT+04] Denger, C., Teranishi, H.: Inspections in Reuse Intensive Software Development Processes Proceedings of the First International Workshop on Quality Assurance in Reuse Contexts. QUARC 2004, 4-10, 2004.

[DR+02] Dunsmore, A., Roper, M., Wood, M.: Further Investigation into the Development and Evaluation of Reading Techniques for OO Code Inspection. Proceedings of the 24th Int. Conf. on Software Engineering, pp. 47–57, 2002.

[DW98] Desmond F., Wills, D'Souza and A. C.: Objects, Components and Frameworks with UML: The Catalysis Approach. Object Technology Series. Addison Wesley, 1998.

[Fag76] Fagan, M. E.: Design and code inspections to reduce errors in program development. IBM Systems Journal, 15(3):182–211, 1976.

[GG93] Gilb, T., Graham, D.: Software Inspection. Addison-Wesley, 1993.

[Gro04] Gross, H. G.: Component-Based Software Testing with UML, Springer, 2004.

[Kru98] Kruchten, P.: The Rational Unified Process. An Introduction. Object Technology Series. Addison-Wesley, 1998.

[Lai00] Laitenberger, O.: Cost-effective Detection of Software Defects through Perspective-based Inspections. PhD thesis, University of Kaiserslautern, 2000.

[MB+87] Mills, H.D., Basili, V.R., Gannon, J.D., Hamlet, R.G.: Principles of Computer Programming: A Mathematical Approach. Allyn and Bacon Inc., 1987.

[OMG] Object Management Group. OMG Unified Modeling Language Specification, Version 1.5.

[TS+99] Travassos, H. G., Shull, F., Fredericks, M., Basili, V.: Detecting Defects in Object Oriented Designs: Using Reading Techniques to Increase Software Quality. Conference on Object-oriented Programming Systems, Languages & Applications (OOPSLA99), 1999.

[Wie02] Wiegers, K. E.: Peer Reviews in Software – A Practical Guide. Addison-Wesley, 2002

MDD Activities at Siemens AG

Example of the Utilization of Model Driven Technologies and Automatic Code Generation Techniques within Various Siemens Business Units

Ricardo Jimenez Serrano

CT SE 2
Siemens AG
Otto-Hahn-Ring 6
D-81730 Munich
ricardo.jimenez @siemens.com

Abstract: Currently many organizations are adopting MDD to describe their software systems. Model based techniques can bring many benefits as an embedded software development process but they also expose the development team to different and new risks. In this document we will describe three case studies of successful utilization of Model Driven Development methodologies and automatic code generation techniques within various Siemens Business Units while summarizing the common best practices to succeed with these new software development technologies.

1 Introduction

It is a well known fact that embedded and high-performance systems are becoming more complex. Development teams are becoming larger and they require a medium that communicates architectural and technical direction. All of these technical challenges arise in the face of increasing market pressure and shorter time-to-market windows. As developers require a higher level of abstraction that help them decompress their problem space and manage complexity, traditional embedded software development techniques (hand-crafting, C code nuggets by experienced artisans) fail in the face of today's challenges.

The Unified Modeling Language (UML) was created to be the standard form of expression for software models. Its integrated set of diagrams were created to ease the process of depicting a particular view of the system which allows the developer to concentrate on the big picture as well as on the most important problems of that view without being distracted by implementation details.

MDD (or MDA, as it is also known) is basically a standard framework for modeling software systems. As well as UML, it was created by the Object Management Group (www.omg.com) as an effort to add more value to the investment in modeling techniques which many companies have made in order to use UML. It describes an approach to software development in which the focus and primary artifacts of development are models instead of programs. Of course, programs are still important but they can be easily derived from models by means of automatic code generation processes. In other words, with the MDD and its allied technologies, UML becomes a sort of executable blueprint -the descriptions, instructions and the code for your system all in one package.

Model-based techniques and procedures can bring many benefits to an embedded software development process but they also expose the development team to different and new risks. We have all heard the complaining from many of those who have already tried to use the UML and MDD techniques and have failed to bring the new development ideas into practice. But you may not have the luxury to keep these new methodologies away from your software development process, as nowadays embedded system's complexity and functionality are reaching such levels that the developer's community is starting to demand new types of development strategies. These strategies show up together most of the time hand-in-hand with modeling technologies.

So the relevant question is not "Is it possible to fail with UML?", because we already know that the answer is yes. The relevant question here is "How can I succeed with the UML?" In this document we will describe three case studies on successful utilization of Model Driven Development methodologies and automatic code generation techniques in various Siemens Business Units while we try to summarize the common best practices to succeed with these new software development technologies.

2 Full Code Generation at Siemens ICM

In order to boost developers' productivity and to adapt their development processes to current telecommunication networks' complexity, several Siemens Business units decided to start using MDD techniques to develop controlling software for commuted network nodes.

Before giving the green light to the development departments, two preliminary assessments were carried out to find out the two following points: a) whether the characteristics of the software functionality to be developed were suitable for MDD and code-generation techniques; and b) what MDD tool matched the software development process requirements best.

Regarding the first question, it was possible to answer ‘yes’ due to the fact that the majority of the MDD tools rely on state-machines to generate code, and that these state-charts depict particularly well the predominant, event-driven, reactive behavior pattern in the telecommunication world. Furthermore, the legacy code was not very large and the application framework was flexible enough to integrate it in the MDD tool. With regards to the second question, we will not mention the name of the tools being used by the team members as we want to avoid this paper turning into a "vendor shoot-out".

The development team was a mid-sized, heterogeneous group of people with deep domain and traditional programming knowledge. However, they lacked system modeling and UML experience. Training was crucial at the beginning of the development process, not only on the MDD tool, but also on topics such as modeling techniques and UML guidelines and best-practices. Physical presence of MDD tool and UML experts was required on the development site so that problem tracking, question answering, and improvised workshops could take place as fast as possible.

A small group of ‘heroes’ was selected from the development teams to advance further in the development process to deal with the largest problems before the rest of the developers. Their activities were closely reviewed to find out how effective the training, mentoring and tools were; how well the cyclic MDD activities (analysis, modeling, generation, graphical debugging, test, integration) were mapped into the existing software development process; etc. After a short while, these selected team members were able to help the rest of the developers avoiding common problems, annoyances, or restrictions that they had already experienced and solved.

The UML had to be tailored and adapted to the system domain. The developers were provided with a list of allowed stereotypes, glossary (actors, component names, interface names, etc), modeling elements to be avoided, naming conventions and basic examples that showed how to model typical domain problems with the UML. UML diagrams and constructs were also classified according to importance for each development role in the team.

The biggest problems were those related to adapting the tool to the development and execution platform so that code could be automatically executed from the tool on the final HW target and offer graphical debugging capabilities. The fact that there were different modelers working in parallel on the same model was also a large problem at the beginning. Although the models were kept under a configuration management tool, an incorrect division of the system in terms of subsystems and components made it almost impossible to work in parallel. Some overall-architectural reviews had to be performed in order to correct it. The heterogeneity of the developers was also a drawback and it turned the project into a multi-paced project. Some parts of the system were developed faster than others and the interdependencies between components developed by different teams made the faster teams have to wait for the slower ones. The problem was partially solved by rearranging the people among the teams, but it still handicapped the project all along the process.

Another problem came along when they realized that the code generation capabilities of the tool were sometimes overkill in that it generated heavier code than they needed. It took a long time to customize the algorithms used by the tool to generate the code and even then some results were not totally satisfactory. The teams also noticed that with the MDA process you may run into development problems due to a mis-match between the code that you thought would be generated, and the code that was actually generated. The origin of this problem was the lack of experience with the tool, since the team was not able to anticipate what code would be generated. Of course as they got used to the tool the problems almost disappeared but in the short term the unfamiliarity with the MDA tool did cost them several minor delays.

Finally, the benefits of having platform-independent models were evident when we had to change the final SW-HW target and work in a different environment (emulator) for a while. Just by changing the code generation and compilation settings the teams were able to work on the emulator without having to manually modify the code.

The project was not finished at the time this document was written, and some studies are planned to determine the productivity increase from the use of these new technologies.

3 Elaborative Modeling at Siemens VDO

In an attempt to maintain increasing software complexity and to solve requirement misconceptions as well as other typical communication problems with customers and within the development teams, several Siemens VDO divisions decided to use UML executable models for a fast and early validation of their software systems.

The idea was to basically rely on formal UML models that served as executable specifications of the system to that was to be built. These platform-independent models are supposed to have a very long lifespan and to be able to survive technology changes easily. They did not substitute textual software requirements but complemented them by providing powerful specification analysis facilities, rapid prototyping capabilities and early validation and verification possibilities. Although code generation for the final target was out of scope due to technological reasons (low-level programming language, implementation very close to the HW layer, very tight real time and memory restrictions, etc.), the executable models not only allowed the teams to have consistent specifications from problem analysis to implementation, but also provided strong motivation and training for the necessary adaptation to systematic, code generated, strongly tool-based, future software development processes.

The creation of the models was carried out by the system architects because of their extensive knowledge of the system domain as well as their familiarity of the system architecture. Training activities on the chosen UML tool and the modeling language were easy to perform because of the small number of people that attended. Since models did not influence the development activities directly and since the major effort on modeling was put at the beginning of the development process, there was not big working overhead for standard developers.

The modeling team was also responsible for maintenance of the model and for changing the models upon demand during the development process - especially to depict changes on the software requirements that the customer wanted to make. It is necessary to emphasize that the models proved to be a very valuable tool for the customer to be able to quickly and effectively introduce new requirements or to change requirements during the development process and before the product was finished. The UML tool provided the possibility to link model elements to typical requirement management tools to keep track of where and how software requirements were addressed and implemented in the architecture.

Specification and description of product families was another aspect where the prototyping models came to be very useful. By having a different prototype for each product of the family it was very easy to show the main characteristics that that particular product offered, and also the main differences in comparison to the rest of the family. The models turned out to be the main elements in the catalog of the department's product specification.

The biggest problem to solve was to overcome the inherent tendency to leave the models behind as the systems evolved. The effort to keep the models in sync with the code was underestimated at the beginning of the project and the modeling team had to be very strict and disciplined in order to not forget to modify the models as the system become more mature. They also underestimated the learning curve required to create executable models with the UML tool and had to continually learn to get used to the paradigm shift from traditional development (a typical example was the action required to initialize the system which is very different if you do it through the MDD tool than if you do it manually on the code).

Current plans analyze the possibility of using executable models extensively on further architectural designs.

4 OSGi Code Generation as Research Project

A very well known strategy to handle complexity and diversity on software development is the usage of components. Component-based software development allows the implementation of a certain kind of functionality on a highly encapsulated software entity called component. An application consists of several components which are only connected through very well defined interfaces. Components cooperate with each other by providing and requiring services to or from other components.

Linking components is the task in which an automatic code generation strategy could be very useful to keep the developer away from programming the glue code which makes component communication possible. According to this approach, the developers would concentrate on the functional concerns of the components while using MDD to generate all of the “scaffolding” code that is necessary to compose an application based on components.

The MDD OGSi sample project was carried out by Siemens CT to evaluate and to demonstrate the advantage of using automatic code generation in a project that uses OSGi as the component framework. Without any detailed OSGi knowledge the developer was able to automatically generate the following software elements:

- Component that implements the OSGi bundle activator
- Code to register services with the OSGi registry
- Code to retrieve required services form the OSGi registry
- Code to access required services in a secure fashion
- Meta files (information about imported and exported packages)
- Build files (compile generated class files an pack them into jar files)
- Further control files

The largest problems were related to the learning curve of the MDD tool because the documentation was very poor and due to budget restrictions we did not have any direct support from the toll vendor. We underestimated the effort needed to master the tool and we really noticed the absence of a tool expert sitting next to us - especially at the beginning of the project. Additionally, the generator was not very stable and its capabilities changed quite often during our project, which lead to a continuous learning process of the brand new features. Also, the lack of OSGi experience only added to our difficulties when having to read the generated code to find out why the system did not compile.

But in the long run, once the development environment was set and we became familiar with the tool the productivity did increase and we could experience the benefits of the automatic code generation.

5 Typical Problems and Common Best Practices

According to our experience and by comparing the three previous chapters it seems that the most typical problems when adopting MDD technologies are related to the following aspects:

- Lack of training and support
- Steep learning curves
- Underestimation of a new technology requirements
- MDD tool capabilities and restrictions
- Lack of long term commitment
- Staff knowledge and experience

In order to minimize these problems the best adoption strategy should comprise of at least the following points:

- Develop an understanding of your own key issues and challenges
- Gain a technical overview of the key process and tool requirements
- Secure a very good training for your practitioners
- Select a reliable and flexible tooling support
- Manage risks through an incremental adoption process
- Make sure your staff is ready for the change

6 Summary

Moving forward to a model based development paradigm can bring many benefits not only in the short and medium term, but also in the long term because it conforms to the requirements needed to successfully face the challenges of the software development of the future. It is, however, a complex and challenging undertaking that requires a considerable effort from the staff and the process. But the typical problems can be mitigated by following several best-practices when adopting the new paradigm.

References

[JS04] Jimenez Serrano, R.: Incorporating UML and MDD into your Software Development process. Siemens CT SE 2, Munich, 2004

[Fo03] Fontana, P.: Managing the Risks of Adopting the UML, 2003

[Pth03] PathFinder Solutions LLC: Accelerating Software Development with MDD, 2003

[St03] Stanley J. Sewal: Executive Justifications for Adopting MDD, 2003

[Mi04] The Middleware Company: Model Driven Development for J2EE Utilizing a Model Driven Architecture Approach, 2004

Entwicklung eingebetteter Softwaresysteme mit Strukturierten Komponenten

Felix Gutbrodt, Michael Wedel

Institut für Automatisierungs- und Softwaretechnik
Universität Stuttgart
Pfaffenwaldring 47
70550 Stuttgart
gutbrodt@ias.uni-stuttgart.de
wedel@ias.uni-stuttgart.de

Abstract: Softwareentwicklung für eingebettete Systeme zeichnet sich heute noch häufig durch den Einsatz strukturierter Programmiersprachen aus. Um die Vorteile der objektorientierten Modellierung mit der Effizienz strukturierter Sprachen zu verbinden, bieten sich Strukturierte Komponenten an. Dieser Aufsatz zeigt das Potenzial Strukturierter Komponenten anhand eines Beispiels aus der Praxis. Außerdem wird ein Werkzeug vorgestellt, mit dem sich eingebettete Softwaresysteme mit Strukturierten Komponenten modellbasiert entwickeln lassen.

1 Einleitung

Die individuelle Neuentwicklung von Software ist äußerst aufwändig und kostenintensiv [Göhn98]. Zur Erhöhung der Qualität und zur Reduzierung der Kosten bietet sich daher die Mehrfachverwendung bereits vorhandener Softwarekomponenten an. Dabei vereinfacht eine modellbasierte Entwicklung die Konstruktion von Anwendungen aus einzelnen Komponenten wie bei einem Baukasten.

Softwarekomponenten werden folgende Merkmale zugrunde gelegt [Göhn98] [Szyp97]: funktionale Geschlossenheit, strukturelle Unabhängigkeit, Anpassbarkeit, Verknüpfbarkeit, Offenheit und Unveränderbarkeit durch Dritte. Zusätzlich werden in [EbGö04a] noch Einmaligkeit und Nebenläufigkeit genannt.

Im PC-Bereich existieren zwar Komponententechnologien wie beispielsweise JavaBeans und COM. Diese sind jedoch in eingebetteten Systemen in der Regel nicht einsetzbar, da sie zu viel Rechenleistung und Speicherplatz in Anspruch nehmen. Dieses Problem lösen Strukturierte Komponenten, da sie leichtgewichtig und für geringen Ressourcenverbrauch ausgelegt sind [EbGö04b]. Sie erfordern weder eine bestimmte Programmiersprache noch eine spezifische Laufzeitumgebung und sind somit nicht auf eine Zielplattform festgelegt.

2 Modellierung und Aufbau Strukturierter Komponenten

Zur Modellierung von Softwaresystemen, welche auf Strukturierten Komponenten basieren, bietet sich die Unified Modeling Language (UML) an. Durch diese etablierte Notation wird die Kombination von Strukturierten Komponenten mit objektorientierter Software bereits in Analyse und Entwurf möglich. Um Strukturierte Komponenten eindeutig zu kennzeichnen, sind diese mit entsprechenden Stereotypen ausgezeichnet. Das Profil „UML-PA“ (UML for Process Automation [FGG+04]) definiert diese Stereotypen, indem es die Erweiterungsmechanismen der UML 2.0 [UML03] nutzt (Abbildung 1).

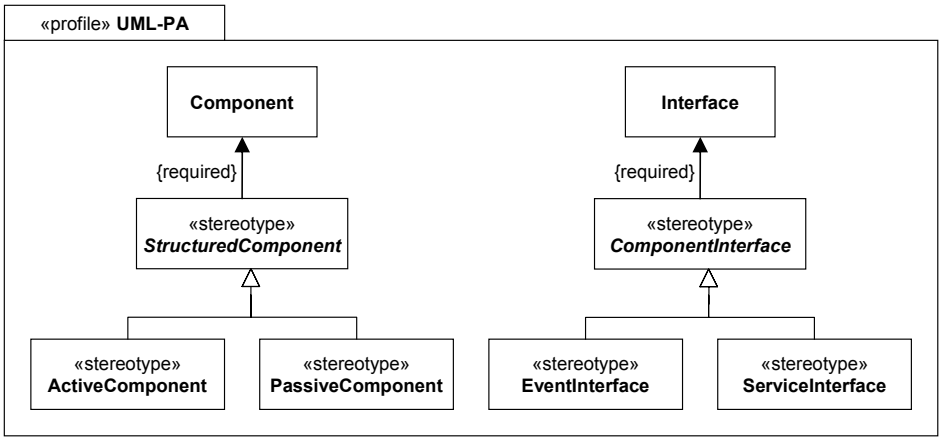

Abbildung 1: Ausschnitt aus dem UML-PA-Profil

Man unterscheidet zwischen passiven Komponenten (Stereotyp «PassiveComponent»), die nur über explizite Aufrufe angesprochen werden können, und aktiven Komponenten (Stereotyp «ActiveComponent»), welche über eine Aktivierungsoperation in regelmäßigen Abständen aktiviert werden. Dadurch kann Nebenläufigkeit in einfacher Weise realisiert werden. Um Echtzeitanforderungen gerecht zu werden, können Strukturierte Komponenten zusammen mit einem Echtzeitbetriebssystem eingesetzt werden. Ihre grundsätzliche Unabhängigkeit von einer spezifischen Plattform erlaubt außerdem das Ausführen und Testen auf einem PC.

Schnittstellen, welche die nach außen sichtbare Funktionalität einer Komponente beschreiben, realisieren die funktionale Geschlossenheit. Dabei wird zwischen Dienstschnittstellen (Stereotyp «ServiceInterface») und Ereignisschnittstellen (Stereotyp «EventInterface») unterschieden. Eine Komponente, welche eine Dienstschnittstelle implementiert, stellt eine gewisse Funktionalität zur Verfügung und fungiert somit als Dienstanbieter. Daher wird eine solche Komponente als Server-Komponente bezeichnet. Umgekehrt kann eine Server-Komponente aber auch ein Ereignis an eine Ereignisschnittstelle melden und somit als Ereignisproduzent auftreten. Analog dazu kann eine Client-Komponente sowohl die Funktion eines Ereigniskonsumenten als auch die eines Dienstbenutzers annehmen (Abbildung 2).

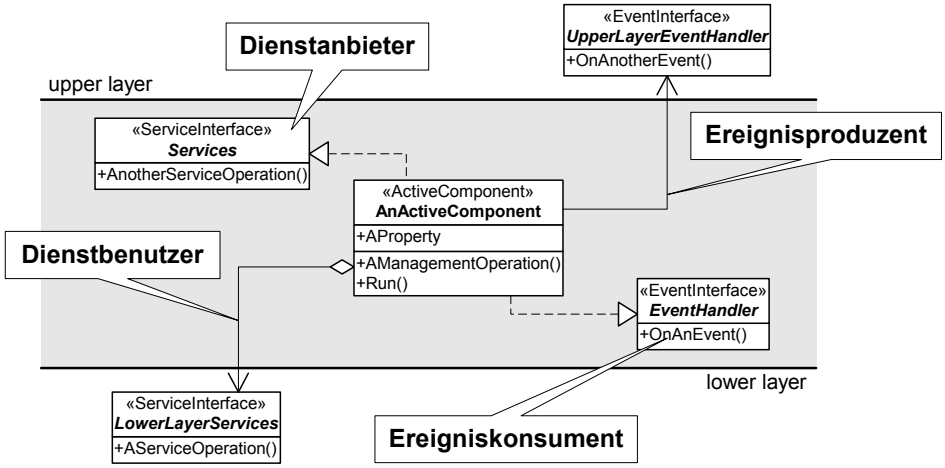

Abbildung 2: Rollen und Schnittstellen Strukturierter Komponenten

Neben den Dienst- und Ereignisschnittstellen verfügen Strukturierte Komponenten über einen weiteren Schnittstellentyp, die Verwaltungsschnittstelle. Wie in Abbildung 2 mit `AManagementOperation` angedeutet, sind dort die Verwaltungsoperationen einer Komponente zusammengefasst. Diese erlauben sowohl die Initialisierung von Komponenten als auch die dynamische Anpassung von Komponentenverknüpfungen. Ebenso ist die bereits genannte Aktivierungsoperation `Run` für aktive Komponenten Teil der Verwaltungsschnittstelle. Darüber hinaus kann eine Strukturierte Komponente mittels der Verwaltungsschnittstelle durch Einstellen von Parametern, Optionen und Eigenschaften konfiguriert werden. In Abbildung 2 ist beispielhaft die Eigenschaft `AProperty` dargestellt.

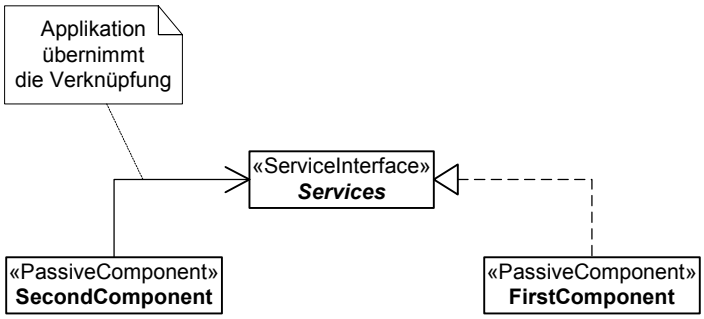

Abbildung 3: Verknüpfung von Komponenten durch die Applikation

Über Dienst- und Ereignisschnittstellen können Komponenten nun miteinander verknüpft werden. Für die Herstellung der Komponentenverknüpfungen ist hierbei die Applikation zuständig, wobei die Verknüpfung sowohl dynamisch zur Laufzeit durch Aufruf von Verwaltungsoperationen als auch statisch zur Compilierzeit erfolgen kann. Um die strukturelle Unabhängigkeit zu realisieren, existieren keine vordefinierten Verknüpfungen zwischen den Komponenten. Somit werden die Komponenten erst von der Applikation, in der sie zum Einsatz kommen, miteinander verschaltet (Abbildung 3).

3 Anwendungsbeispiel

Ein Beispiel für den praktischen Einsatz von Strukturierten Komponenten ist der IAS-WebStack. Hierbei handelt es sich um einen TCP/IP-Protokollstack, der auf die Anwendung in Mikrocontrollern hin optimiert ist. Aufgrund der Schichten-Architektur eignet sich ein solcher Stack sehr gut für die Zerlegung in Komponenten: Die Kommunikation zwischen den Protokollschichten wird auf Schnittstellen von Strukturierten Komponenten abgebildet (Abbildung 2).

Durch die strukturelle Unabhängigkeit der Komponenten lassen sich vorhandene Komponenten zu unterschiedlichen Anwendungen verknüpfen. Als mögliche Anwendung wurde ein HTTP-Server realisiert, wobei eine Komponente Http auf Dienste der Transportschicht [RFC1122] aufbaut. Diese werden durch eine Komponente Tcp zur Verfügung gestellt. In Tabelle 1 ist der Speicherbedarf der Komponenten dargestellt, welche für den HTTP-Server relevant sind.

Name der Komponente[1]	**ROM (Bytes)**	**RAM (Bytes)**	**Code (Bytes)**
Http	453	9	7372
Tcp	113	358	11446
Icmp	12	4	959
Ip	41	30	2314
Arp	42	110	4213
EthDrv	63	8	1412
Lan91c96	35	25	2069
GESAMT	759	544	29785

Tabelle 1: Speicherbedarf ausgewählter Komponenten des IAS-WebStacks

[1] Nach der Namenskonvention der Strukturierten Komponenten für die Programmiersprache C beginnen die Namen von Komponenten mit einem Großbuchstaben, gefolgt von Kleinbuchstaben. Mehrere Worte im Namen können durch weitere Großbuchstaben kenntlich gemacht werden. Beispiel: MyComponentName.

Zusätzlich zu HTTP können auf einfache Weise andere Protokolle der Anwendungsschicht eingesetzt werden. Hierzu kann die Komponente Http durch eine andere Strukturierte Komponente ersetzt oder um eine weitere ergänzt werden, deren Schnittstellen ebenfalls zu denen der Transportschicht kompatibel sind. In Abbildung 4 ist die zusätzliche Integration eines E-Mailclients dargestellt, wobei die Komponenten Http und Smtp mit der Komponente Tcp verknüpft sind. Sowohl HTTP-Server als auch SMTP-Mailclient nutzen so die Dienste der Komponente Tcp. Auf die Entwicklung des grau hinterlegten Teils wird im nächsten Abschnitt eingegangen.

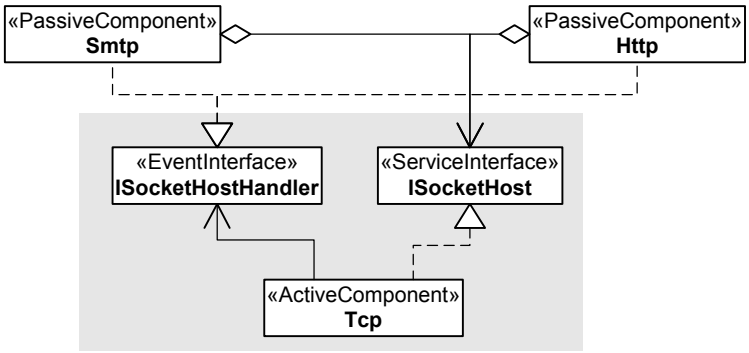

Abbildung 4: Mehrfache Verwendung der Komponente Tcp

4 Werkzeugunterstützung bei der Entwicklung

Wie in Abschnitt 3 beschrieben, erleichtert die Mehrfachverwendbarkeit von Strukturierten Komponenten die Entwicklung von Anwendungen. Um die Unabhängigkeit von einer bestimmten Programmiersprache zu erreichen, geben Abbildungsvorschriften (Idiome) die Transformation in die jeweilige Zielsprache vor. Da im Bereich eingebetteter Systeme die Programmiersprache C vorherrscht, wird in [EbGö04b] ein Idiom für C definiert. Konzept und Idiom geben außerdem die Vorschriften zur Verknüpfung der Komponenten vor. Dieser Vorgang muss bisher für jede neue Anwendung manuell vorgenommen werden. Daher bietet es sich an, die Modellierung in UML und die Erstellung des Quellcodes, der die Verknüpfungen der Komponenten sowie deren Konfiguration festlegt, zu automatisieren.

Damit dies möglich ist, definiert das genannte Idiom für C auch den inneren Aufbau einer Komponente. Dort müssen Datenstrukturen zur strukturell unabhängigen Verknüpfung, Implementierung von Schnittstellen sowie der Komponentenanpassung vorhanden sein. Diese Informationen können ebenfalls modellbasiert erstellt und dann automatisch in Quellcode transformiert werden. Allerdings ist hierzu ein wesentlich höherer Detailgrad erforderlich als für die Erstellung von Anwendungen.

Daher bieten sich zwei unterschiedliche Sichtweisen an: eine auf die Anwendungsentwicklung und eine auf die Komponentenentwicklung. Die Sichtweise auf die Anwendungsentwicklung stellt keine Informationen über den inneren Aufbau der Komponenten dar, gibt aber einen Überblick über verwendete Komponenten und deren Beziehungen und Konfiguration. Dagegen konzentriert sich die Sichtweise für die Komponentenentwicklung auf eine einzige Komponente und deren Schnittstellen. Dem Komponentenentwickler sind sämtliche inneren Details der Komponente zugänglich. Die Entwicklung von Schnittstellen, welche die nach außen sichtbare Funktionalität einer Komponente durch abstrakte Operationen definieren, erfordert denselben Detailgrad.

Hierzu wurde ein Werkzeug entwickelt, das sich als Plugin in die Eclipse-Plattform [Ecli04] einbinden lässt. Eclipse erlaubt das Zusammenfassen verwandter Funktionalität in eine so genannte Perspektive. Dadurch können die Sichtweisen auf Komponenten- und Anwendungsentwicklung in Form von zwei neuen Perspektiven in die Eclipse-Plattform integriert werden.

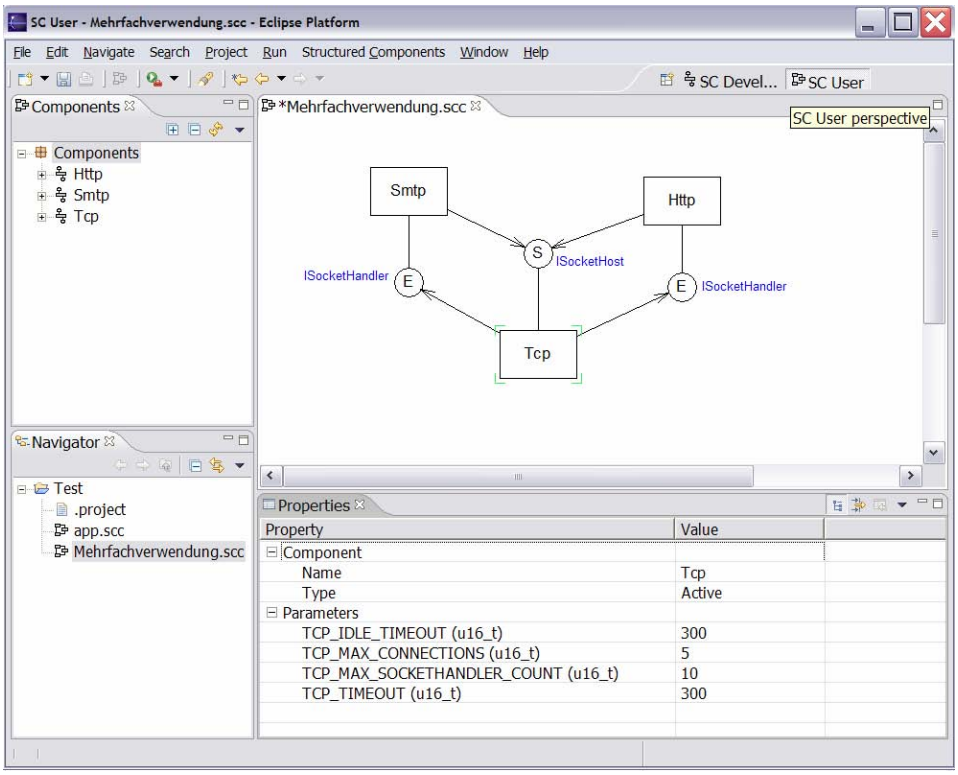

Abbildung 5: Verknüpfung von Komponenten in der Perspektive des Anwendungsentwicklers

Die Perspektive „SC User“ (Abbildung 5) zeigt dem Anwendungsentwickler vorhandene Komponenten an und erlaubt deren Einbindung in die zu entwickelnde Applikation. Zudem lassen sich die Komponenten verknüpfen und konfigurieren. Als Grundlage für die Verknüpfung dienen die Dienst- und Ereignisschnittstellen einer Komponente, wobei nur solche Komponenten miteinander verknüpft werden dürfen, die hinsichtlich ihrer Schnittstellen zusammenpassen. Das Werkzeug liest diese Informationen ein und erlaubt dem Anwendungsentwickler ausschließlich zulässige Verknüpfungen vorzunehmen. Weiter ermöglicht das Werkzeug die Konfiguration der Komponenten über deren Verwaltungsschnittstelle.

Die Notation, die zur Modellierung der Schnittstellen verwendet wird, basiert auf der Lollipop-Notation der UML 2.0 [UML03], um von den für den Anwendungsentwickler irrelevanten Details abstrahieren zu können. Abbildung 5 zeigt, wie das ursprüngliche Modell aus Abbildung 4 in dieser Notation dargestellt wird.

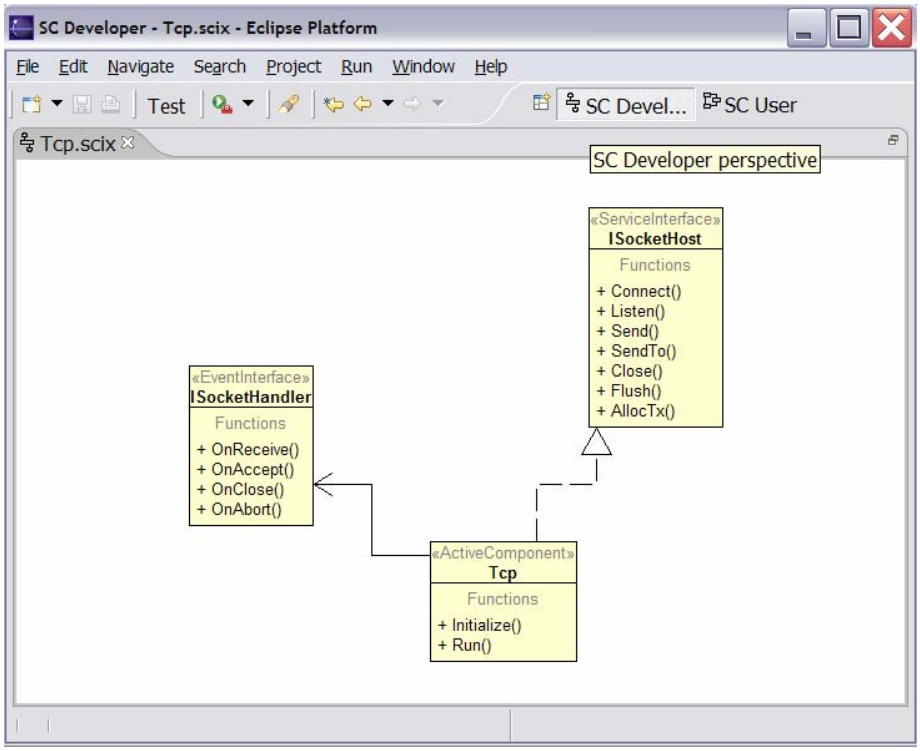

Abbildung 6: Die Perspektive des Komponentenentwicklers

Die Perspektive „SC Developer“ zur Komponentenentwicklung zeigt eine ausführliche Darstellung einer Komponente und der mit ihr in Beziehung stehenden Schnittstellen. In Abbildung 6 ist die Komponente Tcp mit ihren beiden Schnittstellen ISocketHost und ISocketHandler dargestellt. Dies entspricht dem grau hinterlegten Ausschnitt aus Abbildung 4. Es können Operationen hinzugefügt und deren Parameter und Rückgabewerte festgelegt werden. Um die Plattformunabhängigkeit im Modell zu gewährleisten, können alle Modellelemente mit Stereotypen ausgezeichnet werden. Idiome verwenden diese Auszeichnungen als zusätzliche Informationen für die Transformation in die entsprechende Zielsprache. So kann beispielsweise die Länge von Datentypen unabhängig von Zielplattform und Zielsprache vorgegeben werden.

Das Werkzeug macht sich außer der Verwendung von Perspektiven weitere Fähigkeiten der Eclipse-Plattform zu Nutze. So werden eine Versionsverwaltung für die Komponenten sowie ein Hilfesystem integriert. Zudem verwendet es Standard-Ansichten von Eclipse, um z. B. die Parameter von Komponenten darzustellen. Außerdem sieht es die Erweiterung um weitere Quellcodegeneratoren für andere Sprachen über den Extension-Point-Mechanismus von Eclipse vor.

Literaturverzeichnis

[EbGö04a] Eberle, S.; Göhner, P.: Softwareentwicklung für eingebettete Systeme mit Strukturierten Komponenten Teil 1: Komponenten-orientierte Zerlegung. atp – Automatisierungstechnische Praxis 3/2004, Oldenbourg Industrieverlag, München, 2004.

[EbGö04b] Eberle, S.; Göhner, P.: Softwareentwicklung für eingebettete Systeme mit Strukturierten Komponenten Teil 2: Komponenten-orientierte Modellierung und Realisierung. atp Automatisierungstechnische Praxis 4/2004, Oldenbourg Industrieverlag, München, 2004.

[Ecli04] Eclipse Platform Webseite. http://www.eclipse.org, 2004.

[FGG+04] Fischer, K.; Göhner, P.; Gutbrodt, F.; Katzke, U.; Vogel-Heuser, B.: Conceptual Design of an Engineering Model for Product and Plant Automation. Integration of Software Specification Techniques for Applications in Engineering, Springer Verlag, Heidelberg, 2004.

[Göhn98] Göhner, P.: Komponentenbasierte Entwicklung von Automatisierungssystemen. GMA-Kongress, VDI Berichte 1397, VDI Verlag, Düsseldorf, 1998.

[RFC1122] Braden, R. (Hrsg.): Requirements for Internet Hosts – Communication Layers (RFC1122). Internet Engineering Task Force, 1989.

[Szyp97] Szyperski, C.: Component Software - Beyond Object Oriented Programming. Addison-Wesley, Harlow, 1997.

[UML03] UML 2.0 Superstructure Specification, ptc/03-07-06. Object Management Group, Inc., 2003. http://www.omg.org/technology/documents/formal/uml.htm.

Automatic Validation and Verification in a Model-Based Development Process

Dr. Udo Brockmeyer[1], Prof. Dr. Werner Damm[2], Dr. Hardi Hungar[2], Dr. Bernhard Josko[2]

[1] OSC – Embedded Systems AG
Industriestr. 11
26121 Oldenburg
brockmeyer@osc-es.de

[2] Safety Critical Systems
Kuratorium OFFIS e.V.
Escherweg 2
26121 Oldenburg
{damm, hungar, josko}@offis.de

Abstract: This paper describes how a Model-Based Software Engineering Process can ensure correctness of embedded applications. It is shown how formal methods can be used to ensure consistency of the models, and how it can prove that models satisfy selected functional and safety requirements. In addition, it is discussed how to automatically generate test cases from models to verify applications. These two techniques are complemented by automatic code generation techniques.

1 Introduction

Figure 1 taken from a public presentation[1] of Dr. Frischkorn, Head of Electronic System Development, BMW Group, shows the exponential growth in electronic components in cars, spanning all segments from power train, body electronics, active and passive safety, as well as navigation and infotainment. Today, up to 40% of the total vehicle cost originate from the cost of electronic components, with 50%-70% of this share taken by embedded software development. The key strategic role of electronic components in cars is expected to expand. In the close future up to 90% of all innovations are driven by electronic components and software…

This exponential increase in functionality comes together with two other sources of complexity:

[1] ARTIST and NSF Workshop on Automotive Software Development, San Diego, USA, Jan 2004.

- individual functions will be shared across multiple electronic control units (ECUs), i.e. it requires the correct and timely interaction of multiple sub-functions distributed over multiple ECUs. Such distributed hard real-time systems contrast drastically in complexity with the typically single ECU based technologies, for which electronic design processes in place today where optimized. It induces complex interfaces between OEMs and multiple suppliers (in contrast to the traditional situation where one supplier used to provide a complete solution).

- In addition, the introduction of new technologies such as X-by-wire (e.g. "brake-by-wire", such as needed for fully automatic distance control), mandate introductions of new technologies (such as time-triggered architectures), but no yet established design practice exists today.

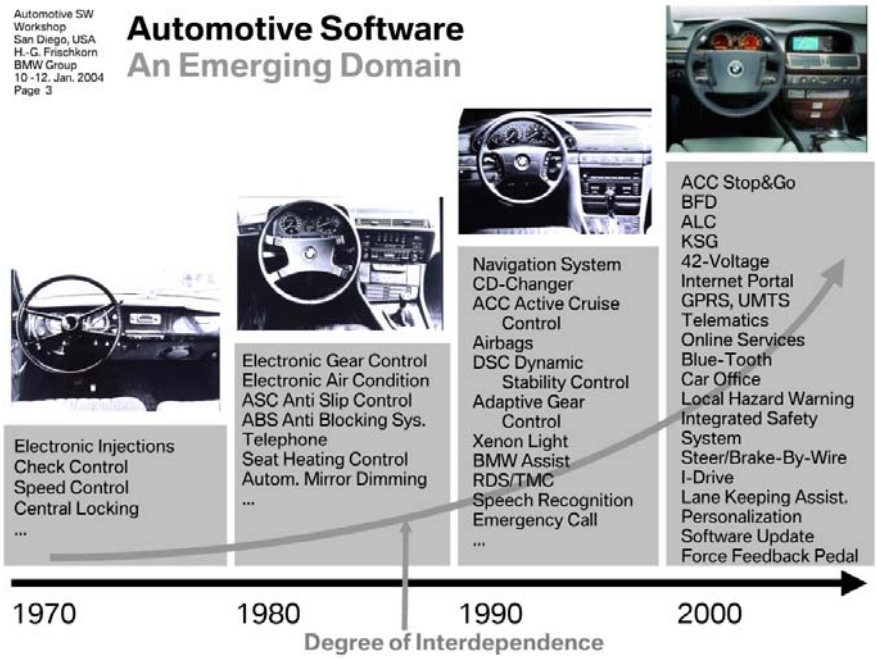

Fig. 1. Automotive Software - An Emerging Domain

Jointly, the above mentioned drastic changes in the development of electronic automotive components have lead to an increased level of failures of electronic components, sometimes making headlines news[2]. As recently reported in "Die Welt"[3], Germany has seen 940 000 cars impacted by call-back actions. The German Automotive Club ADAC reports, that 50 % of all failures treated where due to failures in electronic components.

Multiple lines of attack have been launched by the automotive industry to counter such developments. In particular, there is an increasing trend towards so-called *Model-Based Development*, in which the largely textual way of requirement capturing, still wide-spread today, is replaced by the creation of executable specification models.

Companies have adopted various use cases to capitalize on such specification models. For OEMs, typical use cases include virtual system integration (allowing to simulate the distributed realization of automotive functions prior to contracting suppliers for the development of sub-functions), concept validation for individual sub-functions, assuring that the required functionality can be realized, and the use of specification models as a basis of contracts with suppliers. Typical use-cases on the supplier side include concept validation and automatic code generation, where production quality code is automatically generated from specification models.

Such model-based processes allow a significant improvement of design quality, but *fail to address the testing gap*. Classical approaches to testing fail in being adaptable to the drastic changes in the development of electronic automotive components for inherent reasons. Classical approaches to testing are guided by the test-engineers intuition and experience in designing "good" test cases, test cases, which have a high likelihood in exposing errors. In the classical setting, each ECU development team will have at least two to three test engineers, responsible for designing test cases for acceptance testing on the basis of textual requirement documents and for integration testing. Ideally, test cases cover all functionality, address boundary conditions, are designed to expose typical design faults, and should cover the full ECU code.

---

[2] Spiegel On-Line reported in May 12, 2003, that the finance minister of Thailand was caught in his BMW due to a failure of the central locking system, further impaired by a failing climate control system. PC magazine reported in January 2001, that Ford has acknowledged a software bug in the cruise control chip, causing the car to dash backwards when entering reverse.

[3] Die Welt, 15.1.2004

The exponential increase in complexity and the increasingly distributed nature of functions render the manual construction of good test cases impossible. It is a simple exercise to calculate the number and width of test cases needed to cover today's ECUs. A typical body electronic control unit would have an interface to one or two CAN-bus systems, as well as a number of sensors and actuators directly connected to the ECU. At the bit level representation used in digital processing, this amounts to a typical width of between 50 to 1000 signals. Internally, specification models to such ECUs would typically have some 10 to 30 sub-functions, each with about 20 to 100 key states. Since these state machines run in parallel, this gives in the order of 50^{20} possible state combinations, further compounded by the fact, that state transitions will typically depend on many other internal variables of the controller, which use fixed-point or floating point representations to represent computational objects like brake pressure, acceleration, speed, RPMs, etc. The total number of possible situations of such a specification model exceeds the number of atoms in the universe.

It is this sheer astronomical complexity that in quantitative terms expresses the testing gap - it is simply impossible even for the most qualified test-engineer to come up with sufficiently good and rich test cases. The challenging question is, how in this jungle can test engineers ever hope to find those critical scenarios, which expose the designers errors - and these are bound to be there.

In the following this paper describes some approaches addressing this testing gap.

1.1 Automatic Model Validation

Requirements, usually collected in requirements documents, are translated first in a more formal and executable specification (Simulink/Stateflow models, Rhapsody UML models) representing the system under design. Such specifications are still expressed in an abstract fashion just capturing relevant functional requirements, but still hiding implementation details like physical characteristics of the concrete hardware or software instance, for example, of a prototype. By formally modeling and specifying a design, *Automatic Validation* technologies can be applied to completely validate whether this formal design specification meets its requirements. After some iteration an engineer will deliver a *Reference-Model* that has been proven to fulfill its intended functional and safety-critical requirements. In subsequent stages of the development process a Reference-Model will be used for actual production code generation.

OSC – Embedded Systems offers a suite of tools for performing model-based automatic validation by the formal verification method of model checking for reactive embedded systems designed in order to **validate** requirements on models. Applying automatic validation tools yields well proven Reference-Models, also known as *Golden Devices*.

1.2 Automatic Test Generation

This approach is complemented by the means of *Automatic Test Case Generation* technologies. Automatic test generation applied on Reference-Models yields rich test suites that can be used to **verify** implementations against requirements, in this case against executable models. It bridges the gap between specification models and implementation, by generating executable test sequences, which can be used for Software-in-the-loop or Hardware-in-the-Loop testing. Driven by user-selected coverage criteria, such as covering all states, all transitions, all guards, toggling all outputs, etc, test cases are automatically generated from specification models. This variant is addressing the key use-case of acceptance testing - such as checking, whether an ECU delivered by a supplier is conformant to a specification model - an ECU will only be accepted, if it reacts to all stimuli provided in the test-vectors with the responses also prescribed in the automatically generated test-vectors.

1.3 Automatic Production Code Generation

Automatic Validation and Automatic Test Generation Technologies are complemented by *Automatic Code Generation*. Auto code generation allows a reliable conversion of software designs that are for instance available as Simulink/Stateflow models or Rhapsody models into highly efficient C or C++ code. The correctness of these conversion can be tested by means of simulation and comparison with results from reference simulations, and by the means of auto generated test cases.

Within OSC – Embedded Systems AG and OFFIS the above mentioned technologies have been fully integrated with a range of tools in industrial usage for developing specification models, including ASCET-SD from ETAS, Matlab-Simulink/Stateflow from The MathWorks, TargetLink from dSPACE, Statemate and Rhapsody from I-Logix, and Scade from Esterel Technologies. All tool names are trademarks of the respective companies. We consider this integration of the underlying methods with industry standard CASE tools to be a prerequisite for the introduction of such methods into the industrial design process for electronic control units.

2 Model-Based Design

As motivated above the numbers of electronic devices in modern automobiles increased enormously within the last few years. Not only the raising number of the Embedded Control Units (ECUs) within one automobile is a challenge, but there is also a very strong increase in functionality in every single ECU. These facts lead to an exponential boost of complexity regarding intra- and inter-ECU behavior.

Development of these systems is only manageable if accurate and sophisticated processes are implemented allowing development engineers to deal with this enormous complexity. Those processes provide a means to deliver the devices under hard time and cost constraints.

The *Model-Based Development* process is an approach that allows engineers to graphically specify the behavior of a system and to simulate and execute it in a very early development stage. Tool environments like Matlab®/Simulink®/Stateflow® offered by TheMathworks or Rhapsody® UML offered by I-Logix are wide spread model-based tools to develop applications for different industrial domains such as automotive, aerospace or rail systems.

Once a model-based development process has been established, engineers are able to apply new technologies and tools to enhance and shorten product development cycles, e.g. by introducing *Automatic Model Validation and Automatic Production Code Generation*.

One goal is to improve the V based development process to save development time and effort while preserving or improving the dependability of the developed systems. The methodology makes it easier to understand requirements and increases the correctness of the requirements, the correctness of the design and the code with respect to the requirements. An integration of system-level and design-level modeling tools allows a virtually integrated V-process that is sharpened up to a Y-based process with the required steps at the bottom of the former V being considerably automated (see Figure 2).

Automatic code generation for the complete integrated system with a certified code generator complying to the DO-178B standard eliminates manual code generation and integration as well as unit testing on the Code level. Automatic Validation partially supersedes testing by proving the correctness of the design at multiple levels ranging from subcomponents for creating golden devices up to the overall virtually integrated system. Technically using the generated C code for verification, even Automatic Validation of the target C code is possible, though not required since code generation can always be automatically validated if needed. For a certain C compiler automatic code validation can even be applied to the binary representation, proving the correctness of each translation. Remaining test cases not covered by verification or validation are finally addressed by automatic generation of test cases, which could additionally be used to fortify verification results.

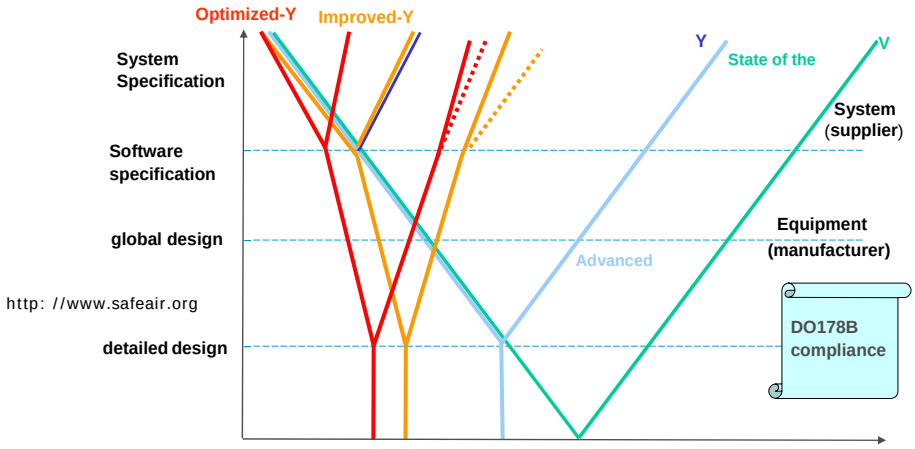

Fig. 2. Improving V based processes. Diagram by Mr. Pilarski, Airbus France

3 Adding Value to Models

3.1 Automatic Model Validation

Automatic Validation by the means of formal verification techniques gives complete coverage of a model with respect to functional, safety and real-time requirements. For example a functional requirement could be: *Intrusion alarm will be activated when window is crashed.* A safety requirement: *Steering wheel will never be locked when ignition is on*. A real-time requirement: *Air bag fires at most 15 milliseconds after a crash*. When verified to satisfy the requirements, the model becomes a “golden device" and can be used as a supplier specification. Later it can be used as a maturity gate prior to sign-off. It significantly reduces the potential for call-backs. Certification can also be used to show compliance to standards imposed by certification authorities for SIL 3,4 applications, such as

- Cenelec EN 50128 - B.30
- DO-178B
- IEC 61508

OSC’s *Statemate ModelChecker*™ and also Simulink/Stateflow *EmbeddedValidator*™ are automatic validation tools where *requirements can be formalized* with the help of a pattern library of typical temporal idioms. It includes patterns like never P, always P, P not before Q, P within t time units after Q etc. Where P and Q are conditions on data items or states of the ECU. The formalized requirements are defined as proofs in a dictionary, where they are maintained for the changes to the model or the requirements.

We have shown certification of Requirements on industrial ECU models within 30 to 300 seconds and executed the Automatic Validation of 36 requirements for an Airbag controller for a total certification in 20 minutes.

Statemate ModelChecker™ and *EmbeddedValidator*™ can also be used to check the *consistency* of e.g. Statemate models automatically. It offers several analysis to debug specification models. It is used for formal debugging and replaces hundreds of simulation runs by one verification run. ModelChecker provides the following types of checks on a model:

- Non-Determinism: The fault when in the model two or more transitions at the same level and with the same source can be fired simultaneously.
- Write-Write Races: The fault when a variable is written with two or more values at the same time.
- Write-Read Races: The possible fault when a variable is written and read at the same time and it is not clear if the old or the new value should be read.
- Range Violation: The fault when a variable can be assigned a value outside it's valid range.
- 'Drive to State': A reach-ability check.
- Drive-To-Property: Checking that certain combinations of variable values can or cannot be reached.

The following are Drive-to-state verification time examples on complete ECU models averaged over all states:

- Autopilot: 30 minutes
- Central locking: 2 minutes
- EMF: 2,5 minutes
- Car-Alarm: 3 minutes
- Airbag: 1 minute

3.2 Automatic Test Generation

If for reasons of cost, space or time an implementation is not generated with a certified code generator, then there is a gap in the formal chain from the model to the actual implemented system. Testing is a way to bridge that gap. But testing is time-consuming, costly and incomplete. Again Automatic Validation can provide an aid. Automatic test generation techniques can be used to automatically generated a full set of test cases from a model. The aim of Automatic Test case Generation (ATG) [DK02] is to automatically generate test cases which cover the entire model. These can then be used for conformance testing e.g. Hardware-In-the-Loop (HIL) and for regression testing (see Figure 3).

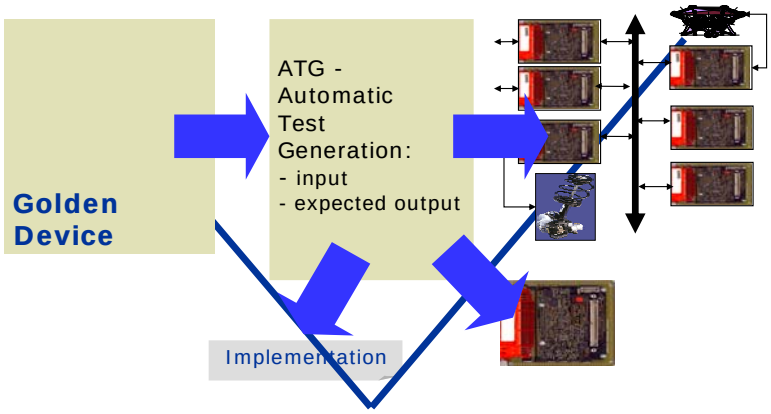

Fig. 3. Automatic Test Generation Technology

Automatic Validation and Automatic Test Generation are complementary.

Automatic Validation provides:

- Formal Verification of functional, safety and timing requirements
- Is purely model-based and abstracts from target Hardware (HW) and Real-Time Operating System (RTOS) and only uses abstract timing
- Gives complete coverage and hence early detection of design errors and integration errors based on a virtual V
- Is fully automatic
- Is used in the early phases of the V

Automatic Test Generation:

Is a Bridge between validated models and real systems in order to perform unit test, ECU test, subsystem integration test and system integration test.

- Testing can take into account real time, distribution and RTOS
- Test cases can be generated automatically
- ATG can re-use the same requirements and models used for Automatic Validation

Testing is approximating formal refinement. Rather than the formal implies, it says that an implementation “Complies to" a model. “Complies to" means that:

- The ECU must have the 'same' interface behavior as its golden device
- 'same': must map model interface objects to ECU interface objects
- 'same': cannot generate complete set of test cases
- Degree of approximation determined by user-selected test objectives

3.3 Automatic Production Code Generation

A production code generator like for instance TargetLink from dSPACE is seamlessly integrated into a CASE tool as Matlab/Simulink. It allows a reliable conversion of software designs that are available as Simulink/Stateflow models into highly efficient C code. Auto code generators take over a lot of responsibilities from the designers and help to deploy more reliable software quicker. The correctness of this conversion can be tested by means of interactive simulation and comparison with results from reference simulations. In order to fully ensure correct behavior the validation of the software design model-based on the generated code is possible. In particular, the combination of automatic test generation technology supports designers in the critical task of production code verification. ATG is applied to generate test cases from a given Reference-Model. These test cases are used to verify the behavior of the auto-generated code that runs on a target system within its context of drivers, software layers to interface to other code parts, etc.

4 Conclusion

A model-based engineering process supported by automatic tools leads to much faster and cheaper development of more reliable automotive applications. Critical tools and technologies to support such sophisticated processes are

- Automatic Validation to formally verify requirements,
- Automatic Test Generation to verify production code, and
- Automatic Code Generation to speed-up the development of more reliable production code.

OFFIS and OSC – Embedded Systems AG are working towards a full process and product solution with respect to the current SW development challenges in the automotive domain and the other domains as trains, chemical, medicine, and clearly aerospace.

References

[DS03] Damm, W., Schulte, C., Segelken, M., Wittke, H., Higgen, U., Eckrich, M.: Formale Verifikation von Ascet Modellen im Rahmen der Entwicklung der Aktivlenkung. Lecuture Notes in Informatics P-34 (2003) 340-345.

[BD01] Baufreton, P., Dupont, F., Lesergent, T., Segelken, M., Brinkmann, H., Strichman, O., Winkelmann, K.: Safeair: Advanced design tools for aircraft systems and airborne software. In: Proceedings of the 2001 International Conference on Dependable Systems and Networks. (2001).

[CG99] Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press,Cambridge, Massachusets, London, England (1999) ISBN 0-262-03270-8.

[HS03] Hunt Jr., W.A., Somenzi, F., eds.: Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, Colorado, USA, July 8 { 12, 2003}, Proceedings. In Hunt Jr., W.A., Somenzi, F., eds.: Computer aided verification (CAV 2003). Lecture Notes in Computer Science, Springer Verlag (2003).

[MP03] Maler, O., Pnueli, A., eds.: Hybrid Systems: Computation and Control (HSCC '03). In Maler, O., Pnueli, A., eds.: Hybrid Systems: Computation and Control (HSCC '03). Lecture Notes in Computer Science, Springer Verlag (2003).

[BD00] Bienmüller, T., Damm, W., Wittke, H.: The STATEMATE verification environment – making it real. In Emerson, E.A., Sistla, A.P., eds.: 12th International Conference on Computer Aided Verification, CAV. Number 1855 in Lecture Notes in Computer Science, Springer-Verlag (2000) 561-567.

[SS90] Stalmarck, G., Sund, M.: Modeling and verifying systems and software in propositional logic. In Daniels, B.K., ed.: Safety of Computer Control Systems (SAFECOMP' 90), Pergamon Press (1990) 31-36.

[B03] Bozzano, M., et al.: Esacs: An integrated methodology for design and safety analysis of complex systems. ESREL (2003).

[BH04] Bretschneider, M., Holberg, H.J., Böde, E., Brückner, I., Peikenkamp, T., Spenke, H.: Model-based safety analysis of a a control system. INCOSE (2004).

[DK02] Bohn, J., Damm, W., Klose, J., Moik, A., Wittke, H.: Modeling and validating train system applications using statemate and live sequence charts. In Ertas, A., Ehrig, H., Krämer, B.J., eds.: Proceedings of the Conference on Integrated Design and Process .Technology (IDPT2002), Society for Design and Process Science (2002).

Early Architecture Exploration with SymTA/S

Kai Richter, Marek Jersak, Rolf Ernst

Institute of Computer and Communication Network Engineering
(Institut für Datentechnik und Kommunikationsnetze, IDA)
Technical University at Braunschweig
Hans-Sommer-Strasse 66
D-38106 Braunschweig
Germany
kai.richter@tu-bs.de

Abstract: With increasingly parallel development of a system's hardware-software architecture on the one hand, and its functionality on the other hand, system integration, verification, and test happens ever later in the design process. In order to ultimately avoid costly re-designs, the system architecture has to more or less meet all requirements on the first try. In other words, the system architects face the challenge to make sufficiently good estimates and choices very early in the design when the implementation is not yet or -in case of re-used or supplied parts- at most partially available. This paper addresses the major limitations of the state-of-the-art benchmarking approach and outlines a structured and systematic architecture evaluation procedure. Based on the SymTA/S tool, the proposed approach explicitly supports estimated data and thereby enables a variety of architectural options to be explored and optimized in early design stages.

1 Introduction

Designing an embedded system is a complex task, and designers face a large variety of serious design challenges. Even before the functions are actually implemented, system architects have to select an appropriate hardware-software architecture out of the large number of available embedded controllers and networks, buses and memories, operating systems and drivers, basic software and libraries, sensors and actuators, etc.. This architecture has to meet a large variety of requirements. Key questions include: Does the communication framework provide the necessary bandwidth? How much bandwidth is necessary? Do the processing units have sufficient computation performance? Can all timing and performance constraints such as end-to-end deadlines be met? Is the power-consumption sufficiently low? Can the system be manufactured at a competitive price? And many more...

Selecting the right components is critical. Over-dimensioning the architecture increases the price and reduces market share. Under-dimensioning the architecture increases the risk of violating performance constraints, thus compromising product quality, and again reducing market share. This shows that the early architectural choices have a dominant impact on the success (or failure) of the project. Essentially, system architects must make sufficiently "good" choices, otherwise the project will simply fail to reach the expected profit.

1.1 How to Make Good Choices Early?

Let us take a brief look at the related field of performance verification. The ITRS [ITRS03] names system level performance verification as one of the top-three IC design issues. The same problem has been recognized by the "AUTOSAR development partnership" (www.autosar.org), in which large parts of the European automotive industry aim at establishing an open standard for automotive E/E architectures. The leading German electronics magazine [Ar04] says "networking and the increasing software complexity pose key challenges on future automotive system design, and requires re-consideration of integration practice, and co-operations".

Today, satisfaction of performance constraints is checked as a side-effect of functional verification and test, which requires the system to be (almost) fully implemented. Consequently, performance verification happens late in the design process. This increases the risk of late architectural changes, which introduce costly delays and can be project-killing in the worst case. Therefore, a key question is what system architects can do to explore system configurations and gather representative data about the quality of alternative choices early, when the implementation is not yet or -in case of re-used or supplied parts- at most partially available? The answer is simple: if detailed data is not available, system architects must use estimations. Later, however, when more detailed about the implementation become available, it must be possible to seamlessly refine the estimation results.

1.2 Possibilities for Early Estimation

We use an example from the automotive industry. An ECU (embedded control unit) supplier such as Bosch, SiemensVDO, Magneti Marelli, etc., wants to evaluate new processor developments from several competitors (e.g. Infineon, Motorola, Texas Instruments, Phillips, ...). Each semiconductor vendor offers a certain core operating at a range of frequencies with a choice of configurable peripherals and coprocessors. The memory structure including cache is also open and configurable.

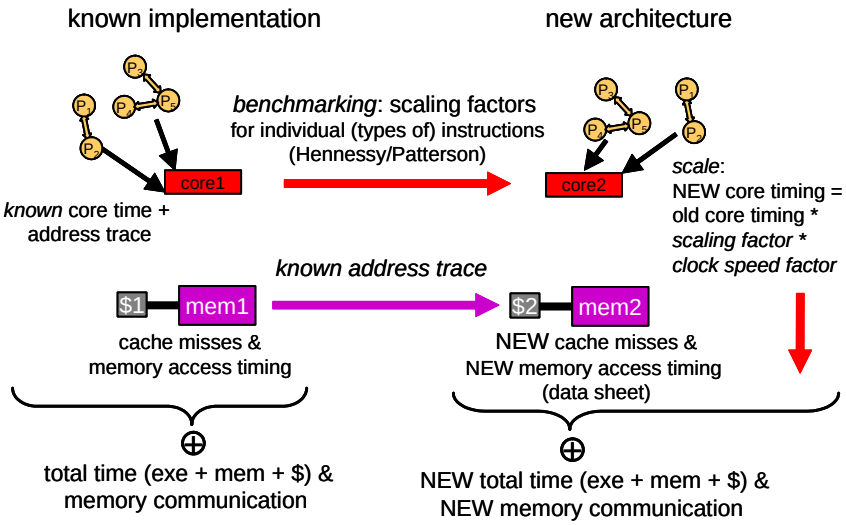

Figure 1 Benchmarking yields timing and memory access patterns of new architecture.

Experienced system architects can project the performance of a known implementation from a previous project to an unknown processor core and new memory configuration. Figure 1 shows how benchmarking helps considerably to derive scaling factors for individual types of instructions or basic functions that allow predicting the timing on the new hardware. In addition, known memory access traces can be re-used and analyzed with new cache hit/miss models to predict the new memory timing.

1.3 Limitations

Unfortunately, benchmarking as described in the previous paragraph is practically limited to simple architectures such as 8 to 16 bit CISC microcontrollers with simple memories. But automotive ECUs are far from simple. For instance, the popular Infineon TriCore and the Motorola MPC555 (see Figure 2) incorporate multiple bridged buses that connect pipelined 32-bit RISC cores to co-processors, caches, partially independent peripherals, and several external memories. Multi-processor ECUs are about to entering the markets. Each ECU, however, forms only one node in today's distributed automotive networks. With such increasing system complexity, the mutual influences due to caching, scheduling, peripherals, bus contention etc. result in ever less reliable estimations and benchmarking is eventually replaced by "guesstimation".

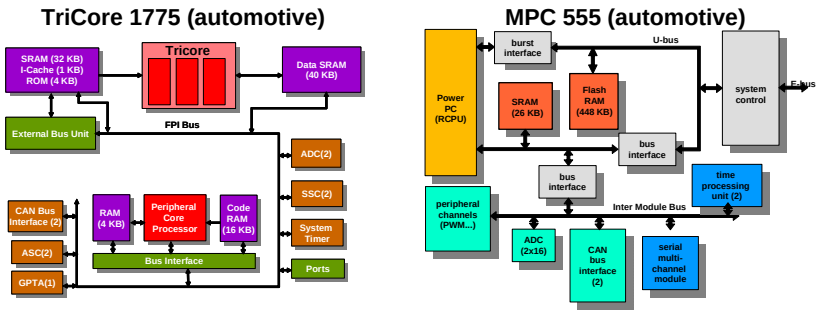

Figure 2 Two complex controllers popularly used in automotive systems.

The problem is in fact a lot worse for automotive OEMs who need to integrate the resulting car platform of heterogeneous ECUs from several competing suppliers, distributed applications with functions partitioned onto several ECUs, and multiple bridged networks running variety of protocols (CAN, LIN, MOST, TTP, Flexray). Furthermore, each car platform has several versions and configurable variants. Is it obvious that benchmarking is not applicable anymore, and even experienced designers can at most roughly guess about the system performance, simply because the variety of dependencies can not be fully overseen by anyone in a design team. Figure 3 illustrates the partitioning or distribution of functions such as automatic cruise control, electronic stability program, and others.

If we look at verification, we can observe similar challenges, even if the entire car platform is fully specified and implemented. Performance simulation and/or test suffer from increasing corner-case coverage problems. The large number of complex performance dependencies leads to corner cases and bottlenecks that are extremely difficult to find and debug, and it is even more difficult to find test patterns to cover them all. In other words, today system-level performance verification has become a second design bottleneck.

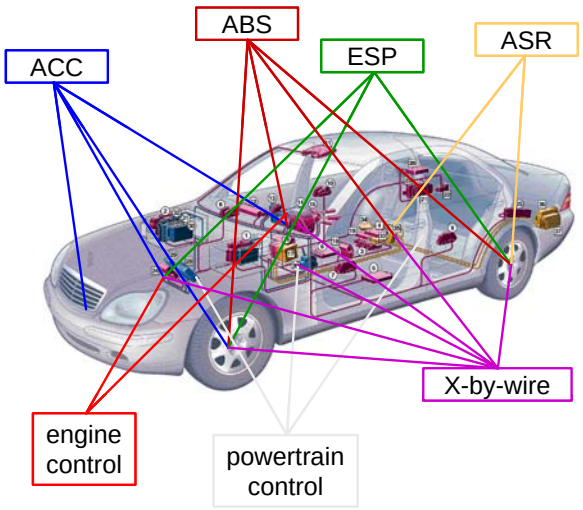

Figure 3 Automotive platforms are heterogeneous, highly integrated, multi-vendor systems.

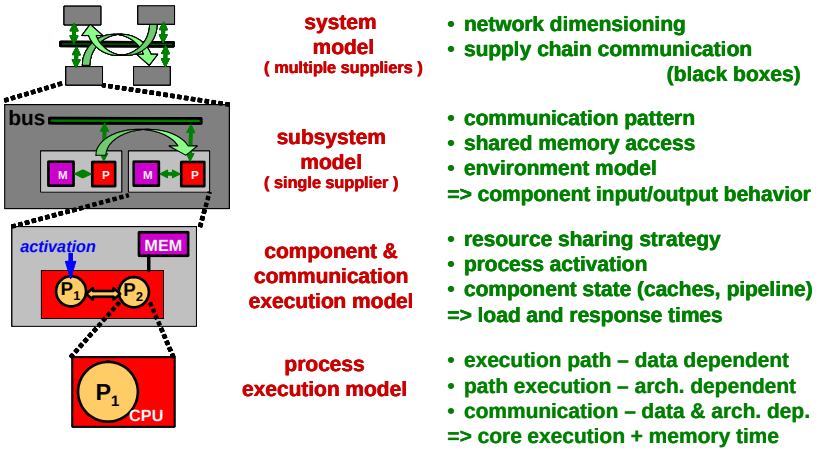

Figure 4 Four structured levels of architecture performance estimation

2 What Else can We Do?

We have seen that the individual pieces of a complex system are manageable in the small, and that platform and system integration is the major source of complexity. This indicates an urgent need for a systematic, structured procedure to handle system performance estimation. We have thoroughly researched this area for several years with a particular focus on practically useful approaches. We recognized that the real-time systems community has developed a variety of formal (i.e. systematic) techniques to structure the entire problem, and concluded that a layered approach is the most promising solution.

Figure 4 shows four architectural levels of complexity. The mentioned benchmarking and projection strategies can be adequately applied at the bottom level where individual task timing and communication is separated from the complex architectural influences. Alternatively, formal WCET (worst-case execution time) analysis can be applied, as proposed by Wolf [Wo02]. AbsInt provides the "aiT" WCET analyzer tool (http://www.absint.com/wcet.htm), that combines abstract interpretation and detailed pipelines models.

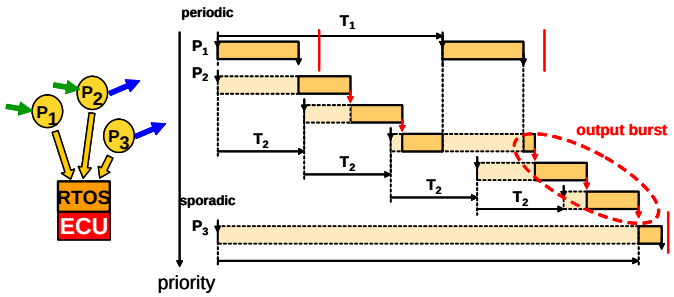

Figure 5 Scheduling diagrams visualize the influence of operating systems on task timing,

The next level (component & communication) already includes mutual dependencies between several tasks including scheduling by an operating system, cache dependencies, and shared-peripheral access. This makes benchmarking less appropriate. Instead, there exist promising approaches, some of them already known for some decades [LL73, JLT85], that use abstract task and activation models and formal analysis methods to determine processor and bus load, task and communication response times, frequencies and jitter, and sometimes the remaining component flexibility. Figure 5 shows the scheduling diagram of a system with three tasks that are scheduled periodically. Scheduling is preemptive and follows static priorities. The highest-priority process P_1 preempts P_2 and P_3, resulting in a complex execution scenario exhibiting jitter and burst process outputs.

Detailed operating system models are starting to become available, and a few real-time analysis tools have already been established, especially in the automotive area. Examples include the Real-Time Architect tool family from LiveDevices (an ETAS Company: http://en.etasgroup.com/products/rta/index.shtml) and Vectors CANalyzer (http://www.vector-cantech.com/products/canalyzer.html). As an additional benefit, these techniques do not require the system to be fully implemented but can also use estimated data, e.g. early estimations of task execution times. This considerably supports system architects during architecture exploration.

For a long time, there was no support for the two remaining, most complex levels in Figure 4, namely subsystem-integration with multiple processors, and system-level integration along the supply-chain. Overseeing the impact of multi-ECU or multi-processor integration that access other peripherals has been a practically unsolved problem requiring detailed I/O patterns to be known to detect overload situations and resolve so-called scheduling anomalies (shown in Figure 6), identify bottlenecks and dimension networks and buffers. The problem is even worse at the system level, where only little internal component details are known due to IP protection. Because of the corner-case coverage problem, neither simulation, nor prototyping, nor test provide sufficient estimation and verification support.

Figure 6 illustrates a so called scheduling anomaly which illustrates the complexity of the overall task of performance analysis and estimation. Recall the P_3 bursts from Figure 5 and consider that P_3's execution time can vary from one execution to the next. There are two corner cases: the minimum execution time for P_3 corresponds to the maximum transient bus load, slowing down other components' communication, and vice versa.

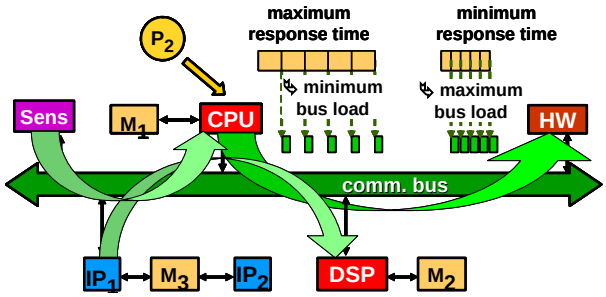

Figure 6 Scheduling anomalies can result from system integration.

3 SymTA/S - A New Technology

We have recently developed a technology and a tool called SymTA/S (http://www.symta.org) that brings approaches from real-time analysis theory to the system level by an intuitive global event flow modeling and analysis technique. SymTA/S does not require fully detailed system specifications but can use estimated data such as task execution times or communication volume. Alternatively, benchmarking, simulation and WCET-analysis can provide more accurate numbers (bottom level in our figure).

At the next (component) level, SymTA/S uses approaches from real-time analysis theory to consider scheduling and arbitration dependencies. Only a small number of parameters such as priorities or time slots are sufficient to provide meaningful information about resource utilization, bandwidth and response times. We have mentioned that tools are available at this level (level 2), but SymTA/S goes much further. It extracts key information from a given schedule and determines the production of system workload, e.g. packets, interrupts, and communication patterns. These influence the global interactions between the components at the system-level (levels 3 and 4), and must essentially be analyzed in order to comprehensively capture the system-level dependencies.

In order to keep track of these dependencies which can usually not be fully overseen by anyone in a design team, SymTA/S uses intuitive workload models or "event stream models" [RE02] that can be used for both scheduling analysis and network analysis. These models capture abstract interaction timing properties such as periods, jitters, and bursts, and provide an adequate understanding of the dynamic system behavior without requiring internal details. Hence the approach is also applicable to black-box integration analysis (level 4).

Figure 7 illustrates the application of event stream models to capture the interaction timing between components in the system, processes P and channels C in the example. We define two classes of models, periodic and sporadic, with three models in each class: strict, with jitter, and with burst. This six-class model set is an efficient compromise between model simplicity and completeness, since these models are sufficient to cover a wide range of systems in practice.

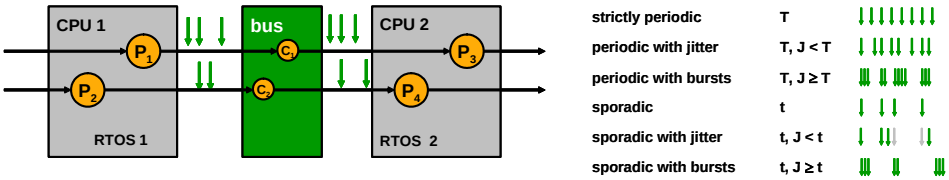

Figure 7 The use of event stream models and their classification.

Controlling the properties of these streams when integrating several tasks and subsystems is key, since they allow system-level performance corner-cases to be found, and bottlenecks to be identified, e.g. overload situations and constraint violations. Based on "event streams", SymTA/S identifies buffer overflows and missed deadlines as a result of transient overload. The influences on other components are automatically detected and propagated further [Ri02].

System architects directly benefit from the ability to use estimations in several places such as task execution times, amount of communicated data, communication patterns, etc... These parameters need not be fixed, since SymTA/S uses interval notations with upper and lower bounds from which the system-level corner cases are systematically derived and analyzed.

Furthermore, SymTA/S explicitly supports the exploration process because it is very flexible with respect to the amount of architectural details. System architects can focus only on their upfront design issues while ignoring unnecessary details such as the pipelining effects or the final bus width. Quite to the contrary, system parameters as well as resource configurations including priorities and mapping of tasks to resources can be changed freely. Since SymTA/S runs extremely fast, it allows evaluating a large number of different architectural choices. SymTA/S supports optimization through a variety of methods that automatically search the design-space for promising solutions based on hard constraints and optimization criteria. The design-space can be freely configured by the user to focus on certain aspects, or to omit certain alternatives because parts of the system have already been fixed. An overview on SymTA/S can be found in [Ha04].

4 Conclusions

Early architecture exploration is a critical task with a huge impact on the success (or failure) of a design project. It requires appropriate estimates of the expected architecture performance for a specific application. The state-of-the-art benchmarking approach, however, can not cope with the increasing complexity of today's systems, and more systematic and structured approaches are needed.

The SymTA/S approach provides this structure, at the same time requiring only few key parameters which can be provided as estimates. For the reasons mentioned above (can use estimated data, allows abstraction from details, supports exploration of alternatives, and provides quick evaluation of architectural changes) SymTA/S is a promising technology for system architects in the early exploration process. And once critical decisions have been made, the SymTA/S specification can be communicated along component supply chains to support the performance verification process throughout the whole design cycle. Models can be refined as new implementation details become available, allowing SymTA/S to verify implementations and detect critical bottlenecks earlier than simulation-based and benchmarking techniques.

We have successfully applied the tool and the technology to several verification and exploration problems in automotive, telecommunications, and multimedia industries where we could detect and solve serious system integration problems. We consider our approach to be a serious alternative to performance simulation and test. The new technology allows comprehensive system integration and provides reliable performance estimates extremely early and with very little computation time.

References

[Ar04] Heinz Arnold. Thema der Woche (topic of the week): Automotive electronics, in German. Markt & Technik, (36):16–20, 2004.

[ITRS03] Semiconductor Industry Association, *International Technology Roadmap for Semiconductors, 2003,* http://public.itrs.net/Files/2003ITRS/Design2003.pdf.

[Wo02] F. Wolf, *Behavioral Intervals in Embedded Software*, Kluwer Academic Publisher, Boston, 2002.

[LL73] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real time environment. *J. ACM*, vol. 20, no. 1, pp. 46-61, 1973.

[JLT85] E. Jensen, C. Locke, and H. Tokuda, "A Time-Driven Scheduling Model for Real-Time Operating Systems, *Proc. 6th IEEE Real-Time Systems Symp*. (RTSS85), IEEE Computer Society Press, Los Alamitos, Calif., 1985, pp. 112-122.

[RE02] K. Richter and R. Ernst, "Event Model Interfaces for Heterogeneous System Analysis," *Proc. Design, Automation and Test in Europe Conf.* (DATE02), IEEE CS Press, Los Alamitos, Calif., 2002, pp. 506-513.

[Ri02] K. Richter, D. Ziegenbein, M. Jersak, R. Ernst, "Model Composition for Scheduling Analysis in Platform Design," *Proc. Design Automation Conf*. (DAC02), ACM Press, New York, 2002, pp. 287-292.

[Ha04] A. Hamann, R. Henia, R. Racu, M. Jersak, K. Richter, R. Ernst. "SymTA/S - Symbolic Timing Analysis for Systems". In *WIP Proc. Euromicro Conference on Real-Time Systems 2004 (ECRTS '04)*, pages 17-20. Catania, Italy, June 2004.

Hardware/Software Co-Synthesis of Real-Time Systems with Approximated Analysis Algorithms

Frank Slomka, Karsten Albers

Department of Computer Science, University of Oldenburg
Ammerländer Heerstraße 114-118, 26111 Oldenburg, Germany
frank.slomka@informatik.uni-oldenburg.de
karsten.albers@informatik.uni-oldenburg.de

Abstract

An important step during the design of embedded systems is to find architectural components and to bind functions (tasks) to this components. The design step is called system level synthesis. The automation of the system level synthesis is limited in recent research by developing models only for standard optimization algorithms.
To improve the design space exploration of embedded real-time systems an new method is described: an adaptive optimization heuristic. The approach is based on a new fully scalable real-time analysis algorithm to shorten the heuristics run-time. A different scaling of the analysis algorithm results in different errors of the analysis results. Controlling the scaling of the analysis algorithms directly by the optimization gives new challenges to design system synthesis applications. The paper also introduces and discusses different techniques of real-time analysis algorithms and compares different approximation algorithms for the feasibility test of real-time systems.

1. Introduction

Future embedded real-time systems will be implemented on one chip. To support the design of such systems it is necessary to develop efficient design methodologies dealing with the hardware and software aspects of the system. The selection of the hardware/software architecture is supported by a design step called system level synthesis. A number of approaches for system synthesis have been proposed in the related literature. The goal is to build an optimization model and to use heuristic optimization techniques to solve the synthesis problem. The optimization problem itself is solved using simulated annealing [2], genetic algorithms [2], [10], [11], [12], [14], [20] and tabu search [2], [19]. In some papers self developed heuristics are presented [8], [9] [15], [18]. Most of the papers considering the analysis of real-time systems, in the meaning that different system tasks with different priorities running on one processor must hold given deadlines.

Optimizing embedded systems is a multiobjective problem. The most important objectives are time, area and power [10], [11], [12], [14], [19], [20].

However, no paper deals with aspects to improve the quality of the optimization heuristic by information coming from the application domain.The results given in [19] are showing the potential of such a technique. Additionally, real-time analysis is a computing intensive task. Therefore the impact by using a scalable real-time analysis algorithm which is controlled directly by the optimization technique is investigated. Additionally it is shown that the adaptive analysis produces better results then previous approximation techniques used in system level synthesis.

* The research described is supported by the *Deutsche Forschungsgemeinschaft* under grant SL 48/ 1-1

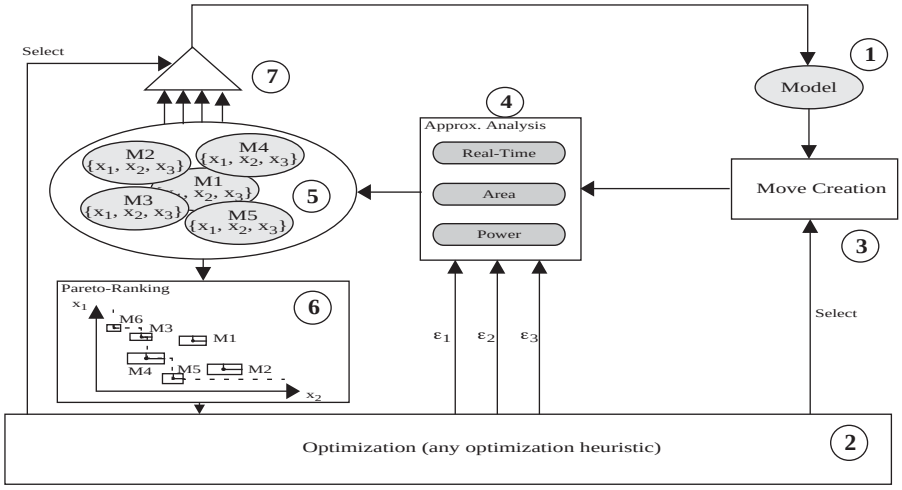

Figure 1: System synthesis with approximative real-time analysis

2. Adaptive System Synthesis

2.1. Structure

For the efficient synthesis of embedded real-time systems we suggest a new coupled procedure with approximated real-time analysis algorithms. The structure of the new approach is given in Fig. 1: As described in Section 2.2. the real-time system is described by a model of communicating tasks with given deadlines (1). For system level synthesis it is possible to bind each task to different hardware components. Binding a task to different components results in different worst case execution times of the tasks. The optimization algorithm (2) generates new solutions from previous by using a move generator (3). This move generator changes the bindings of tasks or the allocation of components.

Each of the new solution candidates must be evaluated concerning the three criteria time, area and power (4). Only candidates keeping the specified real-time deadlines represent valid solutions. It could be differentiated between accurate, approximate and estimating analysis of real-time systems. In this work an approximative real-time analysis algorithm is used, the accuracy of the algorithm can be changed by the optimization heuristic during run-time. Because the run-time of the analysis depends on the accuracy it is possible to switch between fast and inaccurate and slow and accurate analysis. The result of the analysis and the underlying error are then stored together with the available model. Move creation and candidate evaluation is repeated a few time s to build a complete set of candidates called neighborhood, (5) of the first, initial candidate.

All candidates of the neighborhood are then compared against each other using a method called *pareto-ranking* (6) [10]. It is determined whether and how often a candidate is dominated by other candidates. Bad candidates are deleted from the neighborhood and one of the remaining candidates is selected by the optimization algorithm (2) to generate in a further iteration a set of new candidates (7)

2.2. Optimization Model

2.2.1 Event Streams

The event stream model describes the worst case timing relationships between events. The idea is to define a minimal distance in time between one, two, three or more events in a formal specification of input stimuli. The model defines the maximum number of events in different given time intervals. Each event stream consists of a fixed number of event tuples:

$$ET = \binom{p}{a}$$

with a given cycle or period p and an interval a. An event stream is then defined as an ordered set of event tuples with a fixed meaning of each tuple:

$$ES = \{ET_i\} = \left\{\begin{pmatrix} p_i \\ a_i \end{pmatrix}\right\}$$

The meaning of each tuple is defined by its position in the event stream. In this notation $i \in N$ represents the number of events which occur in the interval a_i. This means a_2 is the minimal distance between two events, a_3 the minimal distance between three and a_n is the minimal distance between n events. This sounds confusing, but in this notation the minimal distance between one and one event has to be defined, too.

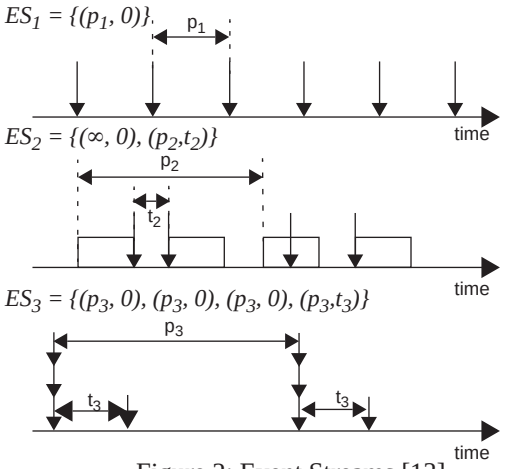

Figure 2: Event Streams [13]

Consider now the examples given in Fig. 2: The first example describes a periodic event as known from standard real-time analysis. The minimal distance between one and one event is *0* and the period of the stimuli is p_1. The second example shows a periodic event that jitters. The box in the figure gives the time interval in which the event jitters. This interval is not important for the analysis algorithm, because an event stream has to describe the worst case of timing relationships of events. However, the minimal distance between two events is t_2. The interval t_2 occurs if an event occur at the end of the jitter interval while the next event is released at the beginning of the jitter interval. The replication period of this event stream is p_2. The last example shows the release of more then one event at one point in time. The minimal distance between one, two and three events is *0*, the minimal distance between four events is t_3. Note that this formalism only considers the worst case scenario. It is not necessary but possible that all three events will occur at the same time. Within this model it is allowed to describe aperiodic and sporadic tasks. In such a case the period of the task is infinite as shown in example two by the first event tuple.

2.2.2 Architecture Model

The architecture model describes the hardware/software architecture of the system. Additionally it specifies all possible bindings of software tasks to hardware components. The model contains two different graphs according to the model given in [5].

However, to support advanced real-time analysis, the data flow graph which describes the application is replaced by an event dependency graph. A event dependency graph is a modified dataflow graph with a different semantic of the edges. In an event dependency graph an edge is annotated by an event stream de-

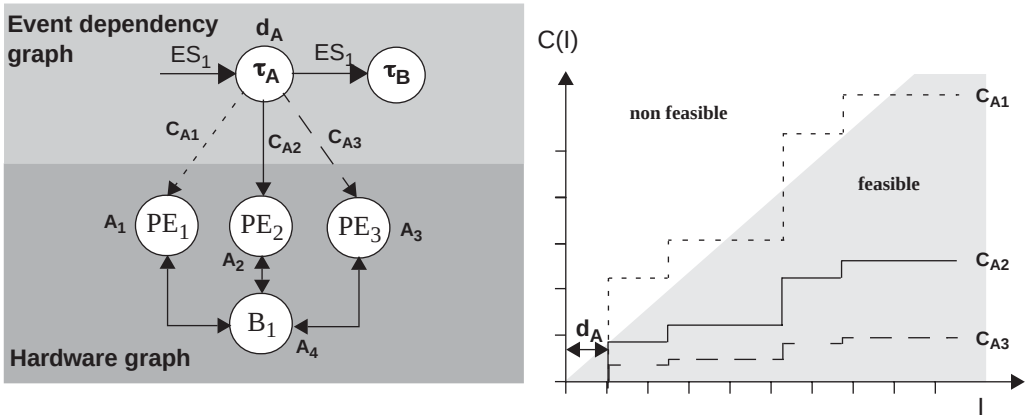

Figure 3: : Synthesis and analysis model

scription. This means that the following task is activated by this event stream. Therefore, the event streams describes the external and the internal triggering of tasks. The advantages of this approach is an easy modelling and analysis of complex input stimuli.

The hardware architecture is described by a hardware graph containing different types of processing elements and communication elements like busses and the interconnection between them.

Each task can be bind to a node of the hardware graph and it is allowed to bind *n* tasks to one hardware component. Possible bindings are specified by an edge from the task to the hardware node annotated by the worst case execution time of the task on this hardware component.

3. Real-Time Analysis

3.1. Demand Bound Function

The algorithm to analyze the real-time behavior of an embedded system depends on the scheduling strategy of the operating system. *Rate Monotonic Scheduling* (*RMS*) and *Deadline Monotonic Scheduling* (*DMS*) are well known strategies to schedule real-time tasks on one processor. Using RMS/DMS as scheduling algorithms leads to a bad utilization of the used processor. A scheduling strategy with a high processor utilization is earliest deadline first (*EDF*). Using *EDF* scheduling the utilization of the processor can be equal to *100%*. This means in terms of scheduling analysis, if the utilization $U \leq 1$ the real-time system satisfies all its deadlines [16].

This result can be used as a starting point to define an analysis algorithm for event streams. The idea of the analysis algorithm is to construct an event function *E(I)* from the event stream and to use this function to build a utilization function of the event stream. The function *E(I)* describes the number of events for a given event stream depending on a given time interval *I*.

If it is assumed that each task has a worst case execution time *c* and a deadline *d*, it is possible to derive the utilization function of an event stream directly from the event function: Each event triggers a task with an execution time *c*. The function *E(I)* is then multiplied with c and moved by d to the right. Such a function represents the workload of a task which must performed by the systems processor within the interval I. After constructing the utilization function of each task triggered by its own event stream, the utilization function of the whole system is constructed by accumulating all separate utilization functions. However, the sum of all utilization functions is also called the *Demand Bound Function* $D_b(I)$ [4] of the whole system.

3.2. Feasibility Test

In *EDF* scheduled systems the condition $U \leq 1$ must hold. This means that the demand bound function of the system shall be always under the bisecting line as shown in Fig. 3. The analysis algorithm given in [13] has to check if $D_b(I)$ is always smaller or equal to the bisecting line for all intervals *I*.

However, about the discontinuous nature of the demand bound function a test algorithm has to check a lot of test points in real life problems. The number of test points depends on the number of events and their periods and can be calculated by

$$I_{\max} = \frac{U}{1-U} \cdot \max_{1 \leq i \leq n}(p_i - d_i)$$

as given in [3]. A system triggered by many different external events coming in adverse period rates leads to a large number of test points. Such a feasibility test is also known as *processor demand test* [3].

Let us consider the problem by discussing a small example: Given are two event streams $E_1 = \{(p_1, 0, d_1, c_1)\}$ and $E_2 = \{(\infty, 0, d_2, c_2), (p_2, t, d_2, c_2)\}$ of Fig. 2. Defining E_1: $d_1=2$, $p_1=4$, $c_1 = 2$ and E_2: $d_2=2000$ $p_2=4000$, $c_2 = 900$, $t = 100$) the number of test points only depends on p_2 and is $I_{max} = (0,775)/(0,225) \cdot (4000 - 2000) = 6889$. So for E_2, 2 test points (at time point 2000 and 6000) are necessary, whereas for E_3 more than 1700 test points are necessary. It is obvious that the complexity of the processor demand test also depends on the different periods of the embedded system.

4. Approximated Real-Time Analysis

It is obvious that calling a NP algorithm in each iteration step of the system synthesis software leads to unacceptable run-time behavior for complex problems. In this section three approximative algorithms solving this problem are discussed.

4.1. Slack Based Approximation

A very fast system synthesis algorithm (*MOGAC*) is given in [10]. The complexity of this approach is also reduced by a real-time analysis approximation. The main idea is to use list-scheduling for a non-preemptive scheduling analysis. The scheduling analysis algorithm is extended to handle multi periodic system stimuli and deadlines. However, the extension to analyze periodic task graphs leads to an analysis approach which has to consider all task combinations in an specific time interval. This time interval is given by the *least common multiplier* (*LCM*) of the different periods which is called the hyperperiod. In some cases a lot of periods must be calculated: If the periods $p_1 = 12\ ms$ and $p_2 = 13\ ms$ are given by the system specification, *MOGACs* real-time analysis has to consider the time interval *156 ms (LCM(12,13))*. The problem to analyze long time intervals is solved by using an approximation: Tightening the periods of the stimuli is a harder requirement to the real-time system. Therefore, if some periods are shorter than in the specification and it is shown that the modified task system is schedulable, the originally task system is feasible, too. In the example p_2 can be tightened to *12 ms* which leads to a *LCM* of *12 ms* instead of the originally *156 ms*.
Although the algorithm is used in non-preemptive systems the idea can be used in real-time analysis to minimize the maximum analysis interval I_{max}. However, the slack base method tightens the real-time requirements of the system resulting in an error of the analysis algorithm. This error can be easily calculated using the demand bound approach: The error of the method is given by the distance of the demand bound function to the bisecting line defining the processors capacity. Because of the properties of the bisecting line the distance $C(I) = I$. If the distance between the demand bound function and the bisecting line $C(I)$ is considered as an error, then I is proportional to the value of the error. Using the slack method, the period is tightened and the slack of the period is equal to the interval $p - p_t$. In this case p_t is the new tightened period. Standardized to the analysis interval, the given error of the method is defined as

$$\varepsilon = \frac{p - p_t}{p} = 1 - \frac{p_t}{p}$$

Consider again the previous example: Reducing the *LCM* of *156 ms* to *12 ms* by tightening $p_2 = 13\ ms$ to *12 ms* gives an error of *7.6 %* to the approximation.

4.2. Chakraborty's Approximation

The algorithm of Chakraborty et al. [7] was designed for an approximated analysis of the recurring real-time task system. In Fig. 4 a simplified version of the algorithm is presented. The maximum test interval I_{max} is divided into a number of test intervals, having a fixed distance K to each other. To guarantee the deadlines, the cost for each test interval is calculated separately and compared with the available capacity of the test interval before - in the exact algorithm the cost of a test interval is compared to its own capacity. If the capacity is sufficient for this test interval, it is also sufficient for all possible test intervals between the last and the actual test interval.

ALGORITHM Chakraborty
INPUT: ListofTasks {e_i}, testLimit
$I_{max} = U/(1-U) \cdot max_{1 \le i \le n}(p_i - d_i)$
// Test interval from [4]
$K = \frac{I_{max}}{testLimit}$
FOR ($t \leftarrow 1$ to $\lfloor I_{max}/K \rfloor + 1$)
IF ($D_b(T, f_{max} + t \cdot K) > f_{max} + (t-1) \cdot K$) THEN
=> not feasible
END FOR
=> feasible

Figure 4: Algorithm Chakraborty [7]

Fig. 4 gives an impression how the algorithm works: At the first test point, the demand of the exact test is *0 T.U.* At the second test point the demand is *1 T.U.* Therefore the approximation generates a demand of *1 T.U.* at the first test point. The error of the approximation is equal to the maximum distance between the approximated demand bound function and the real demand bound function. It is limited by the distance between the test points. Using test points which are a bit more relaxed, e.g. have a bit more distance to each other, the approximated demand at the first test point could be doubled in the example. This is the case if the second test point slips behind the request of the second unit. Then the first test point generates a demand of *2 T.U.* If the test points are more relaxed, the first test point slip behind the first demand, the approximation becomes infeasible. The complexity of this approach is $O(n^2/\varepsilon \cdot [n^3 + \log(n)])$ for exact one task graph.

ALGORITHM Superposition Approximation
INPUT: ListOfEvents {e_i}, testLimit
$U = \sum_i c_i/p_i$
IF U>100%
=> not feasible
$I_{max} = U/(1-U) \cdot max_{1 \le i \le n}(p_i - d_i)$
, // Test interval from [4]
$\forall e_i \in D_b$: testlist.add(f_i, e_i);
WHILE ($I_{act} \le I_{max} \vee Liste \neq \{\}$)
i = testlist.getNextDemand();
I_{act} = testlist.intervallForDemand(i)
$D'_b = D'_b + c_i + (I_{act} - I_{old}) \cdot U_{ready}$
IF ($D'_b > C_b(I)$)
=> not feasible
IF ($I_{act} < (k-1) \cdot p_i + f_i$)
testlist.add($I_{act} + p_i, e_i$)
ELSE
$U_{ready} = U_{ready} + c_i/p_i$
END IF
$I_{old} = I_{act}$
END WHILE
=> feasible

Figure 5: Superposition Approximation [1]

4.3. Approximation by Superposition

The approximation by *Superposition* is introduced in [1].The idea of the algorithm is to limit the number of test points separately for each utilization function by constructing an approximated demand stream element function $D'_{b_i}(I)$ and to superpose then all approximations to a approximated demand bound function $D'_b(I)$.

The number of test points can be reduced easily by linearization of the demand bound function. However, only the linearization of the demand bound function or the utilization functions leads to a bad approximation. A better approximation can be constructed by computing exactly a firm number of test points and to approximate all left test points by linearization. If *k+1* test points are considered the maximal test interval for each element of the event stream is given by

$$I_m(E_i) = k \cdot p_i + d_i$$

If each event stream is considered separately by building a demand stream element function, an approximated demand stream element function $D'_{b_i}(I)$ can be formulated using the maximal test interval:

$$D'_{b_i}(I) = \begin{cases} D_{b_i}(I_m(E_i)) + \dfrac{c_i}{p_i} \cdot (I - I_m(E_i)) & I > I_m(E_i) \\ D_{b_i}(I) & I \le I_m(E_i) \end{cases}$$

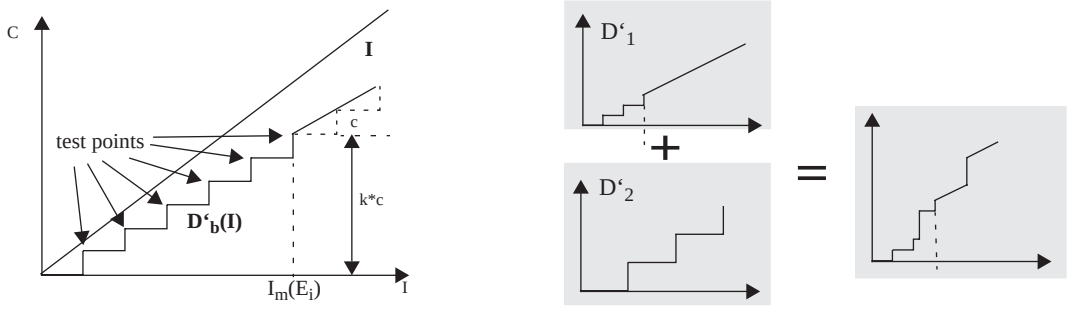

Figure 6: Approximation of the Demand Bound Function by Superposition [1]

As Fig. 6 shows, the approximated demand stream element function is always equal or greater than the demand stream element function. Since $c_i/p_i \leq 1$ it is only necessary to test the points up to $I_m(e_i)$ .The approximated demand bound function is then given by:

$$D'_b(I) = \sum D'_{b_j}(I)$$

The relevant test points of $D'_b(I)$ are all the test points of the elements $D'_{b_i}(I)$. For intervals larger than $I_m(e_i)$ the approximated costs for e_i have to be taken into account at each remaining test interval of the demand stream elements.
The error of the approximation is the difference between $D_b(I)$ and $D'_b(I)$. Therefore the error can be bounded by the test limit *k*. Hence, the ratio between the error and the complete costs is lower than $c_i/k \cdot c_i = 1/k$ in the worst case.
For an efficient testing algorithm it is not necessary to calculate $D_b(I)$ for every test point separately. Fig. 5 shows the complete approximation feasibility test algorithm. It can be shown that the complexity of the approximated feasibility test is $O(n/\varepsilon \cdot \log(n))$.

4.4. Comparing Real-Time Analysis Algorithms

Two case studies are considered as real live examples: The first example is completely described in [6] and models the *Olympus Attitude and Orbital Control System* for Satellites. This example contains 10 periodic and 4 sporadic tasks. The second example was given by Ma and Shin [17]. The model describes a *Palmpilot Application* containing 13 different tasks.
In Tab. 1 the results of using two different variants of the slack based method are given. First a *LCM* of *9000 ms* is considered. In this case the maximal error of the method is *16%* and *11250* test points must be checked by the feasibility test. If a shorter *LCM* of *3000 ms* is given only *3750* test points have to be checked. Then the maximal error is *19,7%*, because of the large slack given by using a period of *150 ms* instead of *187 ms* for task τ_{14}. If this result is directly compared to Chakraborty's approach a maximal error of 0.05% is given by analyzing only 2000 test points. The example shows clearly the disadvantages of the slack based method: Finding the shortest *LCM* with the smallest error is an combinational optimization problem, and results in long computation times to find the optimal slack. Additionally the error of the method is very huge compared directly with other methods.

Task		*Slack LCM = 9000*			*Slack LCM = 3000*		
No.	*Period*	*Period*	*e [%]*	*k*	*Period*	*e [%]*	*k*
τ_1	50	50	0	180	50	0	60
τ_2	10	10	0	900	10	0	300
τ_3	200	200	0	45	200	0	15
τ_4	200	200	0	45	200	0	15
τ_5	200	200	0	45	200	0	15
τ_6	100	100	0	90	200	0	30
τ_7	100	100	0	90	100	0	30
τ_8	200	200	0	45	100	0	15
τ_9	1000	1000	0	9	200	0	3
τ_{10}	1000	1000	0	9	200	0	3
τ_{11}	0.96	0.8	16	11250	0.8	16	3750
τ_{12}	62.5	60	4	150	60	4	50
τ_{13}	100.0	100	0	90	100	0	30
τ_{14}	187.0	180	3.7	50	150	19.7	20

Table 1: Olympus Satellite System: Slack Based Method

The rest of the section concentrates on comparing Chakraborty's algorithm with the superposition method. To perform the experiments and to compare the results directly both algorithms are implemented in Java running on a 1 GHz PowerPC on MacOS X. Each experiment runs 100 times to eliminate the influence of the operating system. The experiment is started with a given error of 20% to 0.01%. For each experiment the number of test points and the minimal distance to the bisecting line as metric for the accuracy is given

Approximation by Superposition					*Approximation by Chakraborty*				
Run-time[ms] (100 runs)	*Approximation*		*Result*	*Min. Distance*	*Run-time[ms] (100 runs)*	*Approximation*		*Result*	*Min. Distance*
	ε [%]	*k*				*ε [%]*	*k*		
262	20	70	sched	0.45	80	20	5	no	-701
434	10	140	sched	0.45	74	10	10	no	-352
730	5	280	sched	0.45	31	5	20	no	-171
1243	2	700	sched	0.45	35	2	50	no	-47
1713	1	1400	sched	0.45	74	1	100	no	-24
2744	0.5	2800	sched	0.45	75	0.5	200	no	-10
4266	0.2	7000	sched	0.45	32	0.2	500	no	-0.99
6518	0.1	14000	sched	0.45	34	0.1	1000	no	-0.27
11423	0.05	28000	sched	0.45	22755	0.05	2000	sched	0.089
22273	0.02	70000	sched	0.45	58081	0.02	5000	sched	0.45
22244	0.01	140000	sched	0.45	116825	0.01	10000	sched	0.45

Table 2: Analysis Results for the Olympus Satellite System

Approximation by Superposition					*Approximation by Chakraborty*				
Run-time[ms] (100 runs)	*Approximation*		*Result*	*Min. Distance*	*Run-time[ms] (100 runs)*	*Approximation*		*Result*	*Min. Distance*
	ε [%]	*k*				*e [%]*	*k*		
1158	20	35	no	-1.3	133	20	5	no	-1960
311	10	70	no	-1.3	96	10	10	no	-975
629	5	140	sched	1.7	19	5	20	no	-474
1519	2	350	sched	2.0	28	2	50	no	-179
2693	1	700	sched	2.0	104	1	100	no	-76
3961	0.5	1400	sched	2.0	51	0.5	200	no	-22
5046	0.2	3500	sched	2.0	7885	0.2	500	sched	0.1
4588	0.1	7000	sched	2.0	16035	0.1	1000	sched	0.1
4999	0.05	14000	sched	2.0	30039	0.05	2000	sched	0.1
5207	0.02	35000	sched	2.0	82156	0.02	5000	sched	0.1
4974	0.01	70000	sched	2.0	181786	0.01	10000	sched	1.1

Table 3: Analysis Results for the Palmpilot Application

in Tab. 2 and Tab. 3. Because of the approximation is conservative it is possible that the test finishes with no result. This means an algorithm means that the task system is not schedulable on the given processor, but it is.

Considering the results of the *Olympus Satellite System* given in Tab. 2. The first correct result given by Chakraborty's algorithm is found by selecting an error of *0.05%*. In this case Chakraborty's algorithm finishes after *227.55 ms* while the superposition algorithm needs only *114.23 ms* for one run. However, the new superposition algorithm allows fast runs in *17.13 ms* assuming an error of just *1%*. In this case it is possible for the superposition approach to explore 10 solutions more than in Chakraborty's test in the same time. However, for a maximal accuracy the test given by Chakraborty needs *1168.25 ms* while the new approach perform a result in *222.44 ms*. In this case the accuracy of both algorithms is the same as seen in the row *Minimal Distance*.

To show the difference between both approaches more clearly, the second example - the *Palmpilot Application for the GPS System* - was modified. In the experiment a new deadline for task τ_7 is assumed. The new deadline is *100 ms* instead of *150 ms* in the original model. Using this modified task set Chakraborty's approximation gives the first result by a defined error of *0.2%* after running *78.85 ms* as the average runtime (Tab. 3). Assuming this error the superposition algorithm finds a result after *50.46 ms*. Introducing an error of *1%* the superposition algorithm finds a result in *26.93 ms*, which is half the time of the *0.2%* result of Chakraborty's approximation. By a given error of *0.01%* Chakraborty's test spend *1817.86 ms* instead of *49.74 ms* the new test uses.
In summary, if we assume a maximal error of *0.1%*, the superposition algorithm is more than *3* times faster than Chakraborty's approximation.

5. Multiobjective Tabu-Search

5.1. Principles of Tabu Search

Tabu search is an heuristic optimization algorithm. Similar to simulated annealing tabu search is based on a neighborhood search. Thus, any new solution is derived from the previous solution. In order to support this the definition of the neighborhood of a solution and the definition of the moves to transform a previous solution to a new solution is of importance.
Different from greedy algorithms, e.g. as gradient search, tabu search also allows moves to solutions with higher cost. This is important to escape from local minima. However, allowing non-improving steps may result in a cyclic search. To avoid cycles, tabu search employs a memory called tabu list. The purpose is to block moves which can lead to cycles. The list could have very different implementations, one way is to store a fixed number of previous moves to avoid their repeat. However, exceptions can exist, e.g. if a solution marked as tabu represents the best solution found so far (aspiration criterion).

5.2. Multiobjective Tabu Search

The idea of pareto ranking can be used to construct an multiobjective tabu search algorithm [19]. Because of tabu search defines moves to construct new solutions and all moves were put to a list, the neighborhood, it is easy to construct a single neighborhood for each objective. The moves are evaluated separately by the single objectives of the neighborhood. Then the separate move list are merged to one list using pareto ranking.
In many cases there are several good candidates in the same neighborhood. A traditional tabu search would consider only the best solution, disregarding the rest. But it could be also interesting in later steps to consider the neighborhood of the second or the third best candidate. Therefore the overall quota of good candidates which are used in the further optimization process can be bad in the simple approach.
On this point other heuristics uses a different way. Genetic algorithms have a pool of solutions and all evaluated solutions are inserted into this pool. The algorithm uses than more or less the good candidates in the pool to generate new solutions. In most approaches there exist a rule to throw the bad solutions out of the pool, some at once, some after several steps. With such a *survival of the fittest* concept the quota of good solutions, solutions which are considered in the further optimization process is much higher than in neighborhood search. But using neighborhood search also has advantages. Considering the complete neighborhood of a move allows to reach better solutions in one step than considering only a few new solutions.
Also neighborhood search has an easy definition of the moves. New moves are generated by changing one or a few parameters of the old solution. In genetic algorithm especially the crossover operator could be rather complex. To generate a new candidates by crossover it is necessary to somehow merge two old candidates. For complex problems, like system synthesis, this merge can become quite complicated, because it is necessary to avoid invalid solutions. Therefore it is hard to find valid *splitting points* for solutions so that the parts can be merged. It is often necessary to use a *repair mechanism* to eliminate invalid solutions. For example [10] defines clusters of candidates and allows crossovers only within or completely between clusters. In both cases different *splitting points* are used. Nevertheless, a additional repair mechanism is needed to avoid invalid solutions [5]. Using this repair mechanism it is unclear how the new generated solution is related to their parent solution or if it is just a kind of random search.

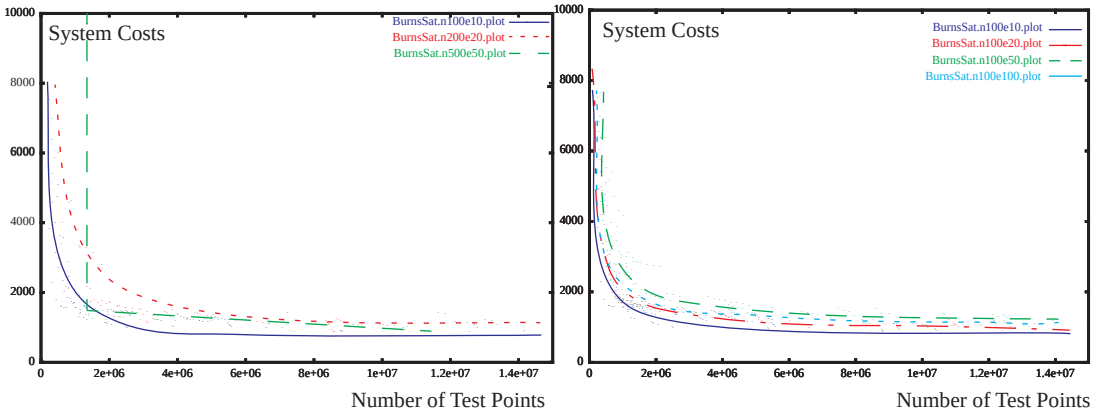

Figure 7: Optimization runs

Considering the evaluation of the different implementations of a system leads to an other problem. In many cases the effort of the evaluation can be reduced using the evaluation of the previous candidate and only considering the differences. This is especially the case if only an inaccurate estimation is used instead of an accurate evaluation. There are often fast possibilities to make a guess for the costs of a new candidate if the costs of related candidates are already known. Candidates which are generated by a simple move and differing only in a few parameters to their parents have a higher degree of similarity than candidates which are generated by a complicated crossover mechanism.

To merge the advantages of both approaches we propose a multiobjective tabu search. First the neighborhood of the start solution is generated and inserted into a pool of candidates. This pool has a limited size. The different objectives of the candidates are evaluated separately. The pool is than sorted with the results of the evaluations using a pareto-ranking. Solutions with an equal ranking are sorted using a priority order. In this step hard constraints are more important than soft constraints. The best candidate in the pool, which is not in the tabu list, is used to generate a new neighborhood. The tabu list contains all solution which have already been considered. To keep the implementation effort low the list only contains the evaluation results for the solutions. Solutions with equal evaluation results are considered as being equal. If the maximum size of the pool is reached, the solutions with the highest rank (most solutions that dominates them) are thrown out. The interesting aspect is the generation of the neighborhood. On this point the optimization is connected with the approximative analysis. A large neighborhood is considered using an large approximation error and therefor an fast evaluation. Out of this presorted neighborhood only the best solution are evaluated with an small approximation error. These best solutions are than considered as the neighborhood and are inserted into the pool.

6. System Synthesis with Approximated Real-Time Analysis Algorithms

Let us now consider the *Olympus Attitude and Orbital Control System* for Satellites. As shown in [1] the task system is schedulable on the processor given in [6]. To use this application in system synthesis it is assumed that a set of different processors are available, with 25%, 50% and 100% of the capacity of the original processor. Additional each task has four instances in the system.

The example is used to study the impact of the adaptive optimization heuristic. To compare different approaches it is assumed that computing each test point of the real-time test needs the same time. As an result the total number of considered test points is proportional to the overall computation time of the algorithm.

In difference to traditional tabu-search approaches the algorithm computes only a limited number of moves of the neighborhood with a real-time feasibility test. Most of the moves are evaluated using the scalable real-time test with a fixed error.

First the fraction of complete evaluated solution candidates to the total number of candidates in the neighborhood is considered. An overview of the results is given in Fig. 7: By running each experiment five times and to accumulate all results in the figure it could be seen that evaluating only 10% of the neighborhood exactly accelerates the optimization and improves the quality of the results.

In contrast to this approach in Fig. 7 the total number of the size of the neighborhood is considered. By using the same fraction of total evaluated to approximated evaluated candidates for all tests it could be seen that a small neighborhood leads to better results.

As it can be seen in Fig. 7 a small number of candidates in the neighborhood is a good choice starting the optimization. Later in the optimization process a large number of candidates brings better results. It looks that using a dynamic neighborhood can improve the optimization results of the algorithm.

A more detailed few to the experiments is given in Tab. 4. The table gives two different kinds of values. In the first columns the constellation is defined. The next block contains the best solution found within an specific limit of test points (1 million, 2 million, ...). The next block shows the number of test points needed to reach a solution better than the limit.

Regard the values in the table. Evaluating only 10% or 20% of the solution exactly leads to less test points necessary to reach the set goals. See for example the needed amount of test points to reach a solution with an better quality than 1500. The trend shows, that evaluating more solutions with a low error needs a higher amount of test points. This is especially the case in the middle or long run, so to reach solution better than 1000 or 900 units. The good run for a neighborhood of 20 can be explained. The last improvement was a big step, from costs of more than 1000 down to costs of 870 in one step. Only if the neighborhood randomly contains this step, this result can be achieved. Only one optimization run contains this step. The other runs with a neighborhood of size 20 have found only solution with costs higher1000 units.

Neighborhood .		Min. Costs archivable reached within number of test points					Number of test point needed to reach a level of costs			
Lenght	Eval	Costs (1 M)	Costs (5 M)	Costs (10M)	Costs (15M)	Costs (20M)	Costs < 3000	Costs < 1500	Costs < 1000	Costs < 900
10	10	1832	1258	--	--	--	426k	2474k	--	--
20	20	1590	1074	870	870	870	385k	3721k	5767k	5767k
100	10	1570	986	870	870	870	342k	1159k	2531k	8805k
100	20	1670	1274	934	892	892	507k	1843k	5359k	12415k
100	50	1810	1512	1080	894	892	622k	3014k	9264k	10436k
100	100	1712	1386	1082	938	892	456k	1426k	12277k	15787k
200	20	2110	1010	870	870	870	805k	1144k	8538k	8575k
200	40	1712	1274	870	870	870	727k	2209k	7462k	11019k
200	100	2778	1158	1010	958	870	889k	3245k	14388k	15185k
200	200	1566	1108	938	870	870	649k	1302k	7837k	14028k
500	50	invalid	1182	870	870	870	1115k	1515k	7325k	7434k
500	500	invalid	1084	1010	870	870	1542k	2304k	10145k	10240k
Simulated Annealing (var. Error)		2052	1768	1736	1572	1572	655k	--	--	--
Simulated Annealing (exact)		2378	1694	1488	1478	1478	748k	9800k	--	--

Table 4: *Optimization results: Olympus Attitude and Orbital Control System* for Satellites

7. Conclusion

In this paper a new approach to solve the problem of system synthesis is presented. Despite to previous approaches it uses an approximative schedulability analysis. It is the first approach where the heuristic optimization algorithm somehow controls the underlying evaluation and made use of the possibility to spend more or less effort, depending on the situation. This paper shows that controlling the effort of evaluation can have an impact and can lead to a more efficient heuristic search algorithm. This effect is not limited to schedulability analysis. Also the analysis of power can be an equal hard part for which also some cheap approximation or estimation algorithm can be used. The proposed optimization algorithm leads to an abstractions and therefor to an separation of the evaluation and the optimization but in a more advanced way than before. As an lesson learned from the results in this paper in future work a dynamic neighborhood length will be considered.

8. References

[1] K. Albers, F. Slomka. *An Event Stream Driven Approximation for the Analysis of Real-Time Systems.* 16th Euromicro Conference On Real-Time Systems. 2004.

[2] J. Axelsson. *Analysis and Synthesis of Heterogeneous Real-Time Systems*. Dissertation, Linköping Studies in Science and Technology, 502. 1997.

[3] S. Baruah, D. Chen, S. Gorinsky, A. Mok. *Generalized Multiframe Tasks.* The International Journal of Time-Critical Computing Systems, 17, 5-22, 1999.

[4] S. Baruah, A. Mok, L. Rosier. *Preemptive Scheduling Hard-Real-Time Sporadic Tasks on One Processor*. Proceedings of the Real-Time Systems Symposium, 182-190, 1990.

[5] T. Blickle, J. Teich, L. Thiele. *System-Level Synthesis Using Evolutionary Algorithms.* Design Automation For Embedded Systems, Kluwer Academic Publisher, Boston, 3(1), 1998

[6] A. Burns, A. Wellings. *HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems*. Elsevier, Oxford, 1995.

[7] S. Chakraborty, S. Künzli, L. Thiele. *Approximate Schedulabilty Analysis.* 23rd IEEE Real-Time Systems Symposium (RTSS), IEEE Press, 159-168, 2002.

[8] B.P. Dave, N.K. Jha. *COHRA. Hardware-Software Cosynthesis on Hierarchical Heterogeneous Distributed Embedded Systems*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. 17(10). 1998.

[9] B.P. Dave, G. Lakshminarayana, N.K. Jha. *COSYN: Hardware-Software Co-Synthesis of Heterogeneous Distributed Embedded Systems.* IEEE Transactions on Very Large Scale Integration (VLSI). 7(1), 1999.

[10] R.P. Dick, N.K. Jha. *MOGAC: A Multiobjective Genetic Algorithm for Hardware-Software Cosynthesis of Distributed Embedded Systems.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. 17(10). 1998.

[11] R. P. Dick and N. K. Jha. *MOCSYN: Multiobjective Corebased Single-Chip System Synthesis*. Proceedings of the Design Automation & Test in Europe Conf., pp. 263-270, Mar. 1999.

[12] R. P. Dick and N. K. Jha. *CORDS: Hardware-Software Co-Synthesis of Reconfigurable Real-Time Distributed Embedded Systems*. Proceedings of the ICCAD. 62-67. 1998.

[13] Gresser, K. 1993. *Echtzeitnachweis ereignisge-steuerter Realzeitsysteme.* (in german) Dissertation, VDI Verlag, Düsseldorf, 10(268), 1993.

[14] C. Haubelt, S. Mostaghim, F. Slomka, J. Teich, A. Tyagi. *Hierarchical Synthesis of Embedded Systems Using Evolutionary Algorithms*. In Evolutionary Algorithms in System Design by Drechsler, R. and Drechsler, N. In Genetic Algorithms and Evolutionary Computation (GENA), Kluwer Academic Publishers, Boston, Dordrecht, London, 63-104, 2003.

[15] C. Lee, ,M. Potkonjak, W. Wolf. *Synthesis of Hard Real-Time Application Specific Systems.* Design Automation for Embedded Systems. Kluwer Academic Publisher, Boston, 4(4), 1999.

[16] C. Liu, J. Layland. *Scheduling Algorithms for Multiprogramming in Hard Real-Time Environments.* Journal of the ACM 20(1), 1973.

[17] C. Ma, K. Shin. *A user-customizable energy-adaptive combined static/dynamic scheduler for mobile communications*. Proceedings of the Real-Time Symposium, 2000.

[18] H. Oh and S. Ha. *Hardware-software Cosynthesis Technique Based on Heterogeneous Multiprocessor Scheduling*. Proceedings of the Int. Workshop on Hardware /Software Co-Design, 183-1878. 1999.

[19] F. Slomka, K. Albers, R. Hofmann. *A Multiobjective Tabu Search Algorithm for Design Space Exploration of Embedded Systems*. IFIP Working Conference on Distributed and Parallel Embedded Systems. Kluwer Academic Publishers, Boston, Dordrecht, London, 2004.

[20] L. Thiele, S. Chakraborty, M. Gries, S. Künzli. *Design Space Exploration of Network Processor Architectures.* First Workshop on Network Processors at the 8th International Symposium on High-Performance Computer Architecture (HPCA8), 30-41, 2002.

UML2-basierte Architekturmodellierung kleiner eingebetteter Systeme – Erfahrungen einer Feldstudie

Alexander Nyßen[1], Horst Lichter[1], Jan Suchotzki[2], Peter Müller[2], Andreas Stelter[3]

1 Einleitung und Motivation

Der neue UML2 Standard bringt eine Reihe von Veränderungen mit sich, die zum Teil, wie die umfangreiche Neustrukturierung des Metamodells, für den Anwender transparent sind, teils jedoch dem Anwender neue Modellierungsmöglichkeiten anbieten. Die wohl bedeutendste Veränderung aus Sicht des Anwenders besteht darin, dass UML2 drei weitere Diagrammtypen (Kompositionsstruktur-, Interaktionsübersichts- und Timing-Diagramme) vorsieht und dass die schon in der ersten Version des Standards enthaltenen Komponenten-, Aktivitäts- und Sequenzdiagramme grundlegend überarbeitet wurden.

Um die neuen Modellierungsmöglichkeiten zu erproben, wurde im Rahmen einer Kooperation zwischen dem ABB Forschungszentrum Ladenburg und dem Lehr- und Forschungsgebiet Software-Konstruktion der RWTH Aachen eine Studie durchgeführt, mit dem Ziel, die Software-Architekturen kleiner eingebetteter Systeme mit Hilfe der UML2 zu modellieren. Dabei konzentrierten wir uns in erster Linie auf die neu eingeführten Kompositionsstrukturdiagramme, die wir exemplarisch an der Modellierung der Software-Architektur von Feldgeräten erprobten. Die dabei erzielten Ergebnisse präsentieren und bewerten wir in diesem Beitrag. Zuvor werden wir jedoch kurz auf die Besonderheiten der in unserer Studie betrachteten Systeme eingehen, damit unsere Modellierungsentscheidungen und das in diesem Beitrag durchgehend verwendete Beispiel besser nachvollzogen werden können.

2 Charakterisierung der betrachteten Systeme

In der hier beschriebenen Studie haben wir die Software-Architektur von Messumformern zur Temperatur- und Druckmessung betrachtet; Messumformer sind Messgeräte, die ein analoges Eingangssignal in ein analoges Ausgangssignal „umformen“ (in unserem Fall ist das Ausgangssignal die Stromstärke eines analogen Ausgangs, des so genannten Stromausgangs, die im Bereich zwischen 4-20mA geregelt wird). Sie zählen zu der Art von Geräten, die, um sie von den Geräten der Steuerungs- oder Leitebene abzugrenzen, als Feldgeräte bezeichnet werden, weil sie im Kontext einer größeren automatisierungstechnischen Anlage „im Feld“, das heißt in direktem Kontakt mit den zu steuernden beziehungsweise zu messenden physikalischen Vorgängen eingesetzt werden. Hardwareseitig zeichnen sich Feldgeräte besonders durch massive Ressourcenbeschränkungen bezüglich Speicherplatz (üblicherweise zwischen 0,5 und 64 KByte RAM und 32-256 KByte ROM), Stromverbrauch und Rechenzeit aus, die sich durch die speziellen Einsatzgebiete im industriellen Umfeld ergeben.

Die Software solcher Geräte ist im Grunde genommen wenig komplex, wenn sich auch hohe Anforderungen durch die starken hardwareseitigen Ressourcenbeschränkungen ergeben. Ihre Architektur wird üblicherweise auf zwei Ebenen beschrieben.

- Auf der oberen Ebene, der *System-Ebene*, werden die strukturellen Abhängigkeiten zwischen Subsystemen, die die zentralen Architekturbausteine des Systems darstellen, modelliert.

- Auf der *Subsystem-Ebene* wird die interne Dekomposition jedes einzelnen Subsystems im Detail beschrieben.

Auf der Subsystem-Ebene wurden bislang typischerweise Klassen- und Objekt- bzw. Kollaborationsdiagramme eingesetzt, während auf der System-Ebene häufig Paket- bzw. Komponentendiagramme verwendet werden ([NMS04]). Diese Diagrammarten tragen einer speziellen Charakteristik der von uns betrachteten Software-Systeme nur unzureichend Rechnung: Die Laufzeitstruktur der Systeme ist relativ starr, das heißt praktisch alle benötigten Objekte (mit Ausnahme einiger Übergabeparameter) werden beim Systemstart erzeugt und existieren über die gesamte Laufzeit. Es wäre daher wünschenswert diese Laufzeitstruktur direkt modellieren zu können, was mit keinem der oben genannten Diagrammarten zufrieden stellend geleistet werden kann. Während Klassen-, Komponenten- und Paketdiagramme nur statische Strukturen beschreiben können, bieten Objekt- und Kollaborationsdiagramme keine ausreichende Ausdruckskraft, um auch komplexe architektonische Abhängigkeiten beschreiben zu können.

Für uns lag daher ein wesentlicher Schwerpunkt der durchgeführten Studie darin, zu untersuchen, ob die im UML2 Standard neu eingeführten Kompositionsstrukturdiagramme diese Lücke schließen können. Nachfolgend werden wir diese kurz einführen, ehe wir die Erfahrungen sowohl beim Modellieren von einzelnen Subsystemen als auch der Gesamtarchitektur im Einzelnen erörtern.

3 Kompositionsstrukturdiagramme

Zusätzlich zu den bereits in der ersten Version des UML Standards enthaltenen Strukturdiagrammen (Paket-, Klassen-, Objekt-, Komponenten-, und Verteilungsdiagramm), die entweder nur rein statische Strukturen abbilden oder eine recht eingeschränkte Ausdruckskraft besitzen, führt der UML2 Standard den Diagrammtyp des Kompositionsstrukturdiagramms (*Composite Structure Diagram*) ein. Dieses erlaubt es, die interne Struktur eines Classifiers in Form von kooperierenden *Laufzeitinstanzen* zu modellieren und seine externen Schnittstellen mit Hilfe von *Ports* explizit zu spezifizieren.

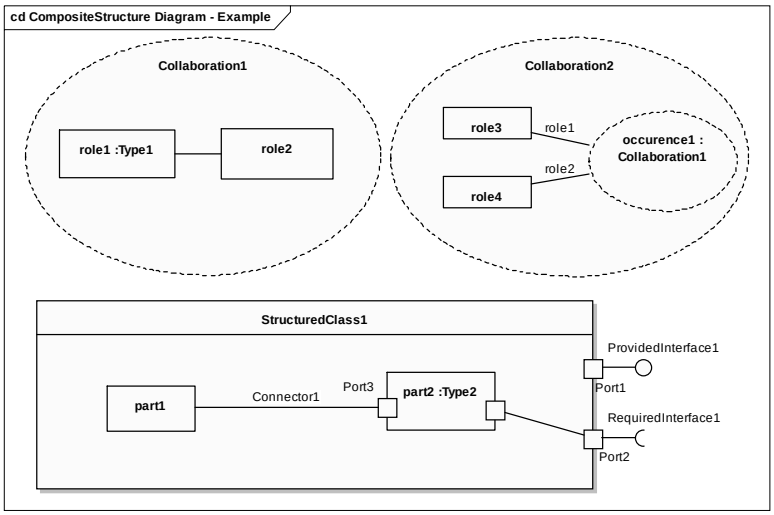

Abbildung 1: Kompositionsstrukturdiagramme verschiedener Ausprägungen

Kompositionsstrukturdiagramme werden wie in Abbildung 1 dargestellt, in zwei verschiedenen Ausprägungen (mit zwar ähnlichen Konzepten aber jeweils unterschiedlichen Notationselementen) eingesetzt:

- für Kollaborationstypen (*Collaboration*) und Kollaborationen (*CollaborationOccurence*)
- für strukturierte Klassen (*StructuredClasses*), Komponenten (*Components*) und Knoten (*Nodes*) in Verteilungsdiagrammen

Während Kompositionsstrukturdiagramme für Kollaborationstypen bzw. Kollaborationen dazu dienen, abstrakte Kollaborationen mit Hilfe von Rollen zu beschreiben, werden solche in der Ausprägung für strukturierte Klassen eingesetzt, um konkrete Klassen (bzw. Komponenten oder Knoten) zu beschreiben, und sind daher bei der Modellierung von konkreten Software-Architekturen in erster Linie von Interesse.

Sie bieten die Möglichkeit, die interne Struktur eines umgebenden Classifiers (Klasse, Komponente oder Knoten) mit Hilfe von internen Bestandteilen (*Parts*) und Konnektoren (*Connectors*) sowie die externen Schnittstellen in Form von Ports (*Ports*) und hervorgehobenen Schnittstellen (*Exposed Interfaces*) zu beschreiben (siehe auch [J04] für weite Informationen):

- Ports beschreiben die Interaktionsmöglichkeiten eines Classifiers gegenüber seiner Umgebung. Sie können beliebig viele angebotene und benötige Schnittstellen (*provided* und *required Interfaces*) aggregieren. Diese müssen nicht unbedingt, können aber in der so genannten *Lollipop*-Notation explizit graphisch dargestellt werden. In diesem Fall spricht man von hervorgehobenen Schnittstellen (*Exposed Interfaces*).

- Parts repräsentieren eine Menge von Instanzen eines anderen Classifiers (zum Beispiel einer Klasse oder Komponente) im Kontext des umgebenden Classifiers. Sie können, wie in Abbildung 1 dargestellt, direkt (wie im Fall von `part1`) oder über Ports (wie im Fall von `part2`), mit anderen Parts oder mit Ports des umgebenden Classifiers mit Hilfe von Konnektoren kommunizieren (die Ports eines Parts sind dabei Referenzen auf die Ports des typisierenden Classifiers). Dadurch kann modelliert werden, wie das Verhalten eines Classifiers durch das Zusammenwirken seiner Bestandteile erzeugt wird.

Da der Grad der Formalisierung eines solchen Kompositionsstrukturdiagramms von relativ lose gekoppelten und relativ unscharf spezifizierten Bestandteilen (Parts ohne Angabe von Typ oder Kardinalität) bis hin zu vollständig gekapselten und präzise beschriebenen Bestandteilen (Parts, die ausschließlich über Ports mit ihrer Umgebung kommunizieren) variieren kann, lassen sich Kompositionsstrukturdiagramme flexibel einsetzen. Ihre Modellierungselemente lassen sich darüber hinaus auch in anderen Strukturdiagrammen (z.B. Klassen- und Komponentendiagrammen) einsetzen, so dass sich auch dort zusätzliche Modellierungsmöglichkeiten ergeben.

4 Modellieren von Subsystemen

Während, wie bereits gesagt, auf der System-Ebene die strukturellen Abhängigkeiten der einzelnen Subsysteme eines Gerätes modelliert werden, wird auf der Subsystem-Ebene die interne Dekomposition jedes einzelnen Subsystems im Detail beschrieben. Wir haben die Eignung von Kompositionsstrukturdiagrammen in beiden Fällen untersucht und wollen die Ergebnisse, zunächst für die Subsystem-Ebene, später für die Modellierung des Gesamtsystems nachfolgend detailliert beschreiben.

4.1 Spezifizieren von Kontextabhängigkeiten

Subsysteme sind die zentralen Architekturbausteine, da sie in sich abgeschlossene und klar abgrenzbare Einheiten darstellen. Damit sind sie die Architekturbausteine, die zum Beispiel für die Software eines Nachfolgegerätes oder die anderer Geräte innerhalb einer Produktfamilie wieder verwendet werden können. Um die Wiederverwendbarkeit zu erhöhen, ist es notwendig, die Subsysteme so zu entwerfen, dass sie in verschiedenen Kontexten ohne großen Anpassungsaufwand einsetzbar sind. Deshalb müssen möglichst alle Kontextabhängigkeiten eines Subsystems explizit beschrieben werden.

Dies gilt sowohl für Abhängigkeiten, die von Subsystemen der Umgebung zu dem betrachteten Subsystem bestehen (also für die Funktionalitäten, die das Subsystem seiner Umgebung anbietet), als auch für Abhängigkeiten des Subsystems gegenüber seinem umgebenden Kontext (also die von anderen Subsystemen erbrachte, vom Subsystem benötigte Funktionalität).

Wie im einführenden Abschnitt beschrieben, bietet die UML2 zur Modellierung solcher Kontextabhängigkeiten in Kompositionsstrukturdiagrammen Ports und Exposed Interfaces an. Da ein Port beliebig viele benötigte und angebotene Schnittstellen definieren kann (die nicht unbedingt auch graphisch in der Lollipop- und Socket-Notation als Exposed Interfaces repräsentiert werden müssen), und ein Subsystem beliebig viele Ports besitzen darf, lassen sich die Kontextabhängigkeiten eines Subsystems explizit beschreiben sowie nach bestimmten Kriterien übersichtlich klassifizieren.

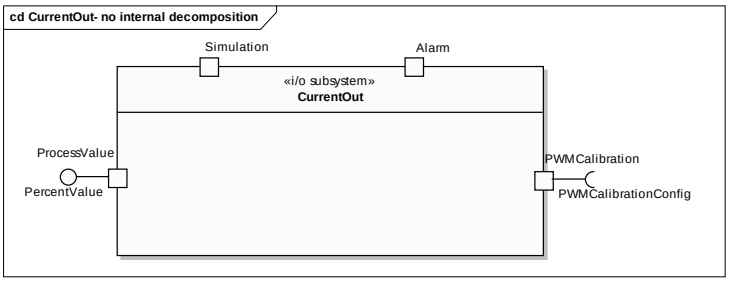

Abbildung 2: Kontextabhängigkeiten des Subsystems `CurrentOut`

Abbildung 2 zeigt dies am Beispiel des Subsystems `CurrentOut`. Seine Funktionalität besteht darin, die Stromstärke des hardwareseitigen Stromausgangs des Messumformers mittels Pulsweitenmodulation (PWM) in einem Bereich von 4-20mA zu steuern, wobei die exakte Stromstärke über einen Prozentwert (0-100%) von der Umgebung gesetzt werden kann. Dies geschieht über das Interface `PercentValue`. Für eine exakte Ansteuerung der Pulsweitenmodulation benötigt das Subsystem Kalibrierdaten, die Aufschluss über die Charakteristika der anzusteuernden Hardware geben. Diese werden von der Umgebung über das benötigte Interface `PWMCalibrationConfig` abgefragt. Die übrigen Ports des Subsystems beschreiben zusätzliche Funktionalität, nämlich die Fähigkeit zur Simulation oder zur Generierung von Alarmen. Sie besitzen Interfaces, die nicht explizit graphisch modelliert wurden.

Neben der reinen Statik der Schnittstellen eines Subsystems, die mit Ports und Interfaces gut beschrieben werden kann, ist aus einer architektonischen Sicht auch von Interesse, wie die Interaktion mit dem Subsystem über seine Schnittstellen bzw. Ports geschieht.

Hier bietet die UML2 die Möglichkeit, für jeden Port einzeln oder aber auch für das Subsystem als Ganzes mit Hilfe eines Protokoll-Zustandsautomaten (*ProtocolStateMachine*) zu modellieren, wie das Subsystem mit der Umgebung bzw. die Umgebung mit dem Subsystem über dessen Ports interagiert.

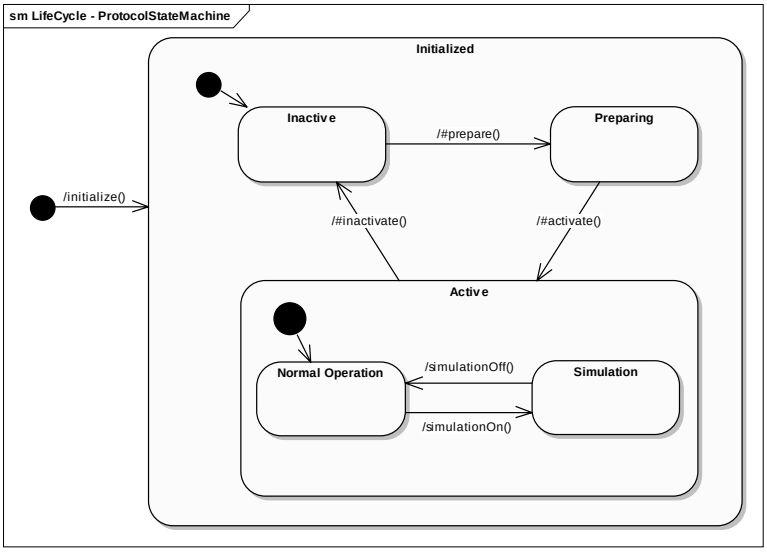

Abbildung 3: Protokoll-Zustandsautomat für den Lebenszyklus des `CurrentOut` Subsystems

Abbildung 3 veranschaulicht dies anhand des Lebenszyklus des in Abbildung 2 dargestellten `CurrentOut` Subsystems. Es zeigt, dass das Subsystem sich nach seiner Aktivierung im Modus „Normal Operation“ befindet und dass die Methoden `simulationOn` und `simulationOff` des Ports `Simulation` nur abhängig vom Zustand des Subsystems aufgerufen werden können.

4.2 Modellieren der (Laufzeit)-Struktur

Die Kontextabhängigkeiten der Subsysteme, d.h. ihre Außensichten, müssen auf der Systemebene beachtet werden, um daraus ein Gesamtsystem zu komponieren. Weiterhin ist für den Entwurf aber auch die auch die innere Struktur der Subsysteme von Interesse, da hier festgelegt wird, wie die vom Subsystem zur Verfügung gestellten Funktionalitäten intern erbracht werden.

Wie bereits ausgeführt, kann mit einem Kompositionsstrukturdiagramm, die innere Struktur eines Subsystems in Form von Parts und Konnektoren modelliert werden. Da Parts eine Menge von Instanzen (im Kontext eines umgebenden *Classifiers*; hier: des Subsystems) repräsentieren, lässt sich so, anders als zum Beispiel bei Klassendiagrammen, die nur die statische Struktur zwischen Klassen abbilden, die Struktur eines Subsystems zur Laufzeit detailliert beschreiben. Dies ist vor allem bei den von uns betrachteten Geräten von großem Nutzen, da sich deren recht statische Laufzeitstruktur damit gut abbilden lässt.

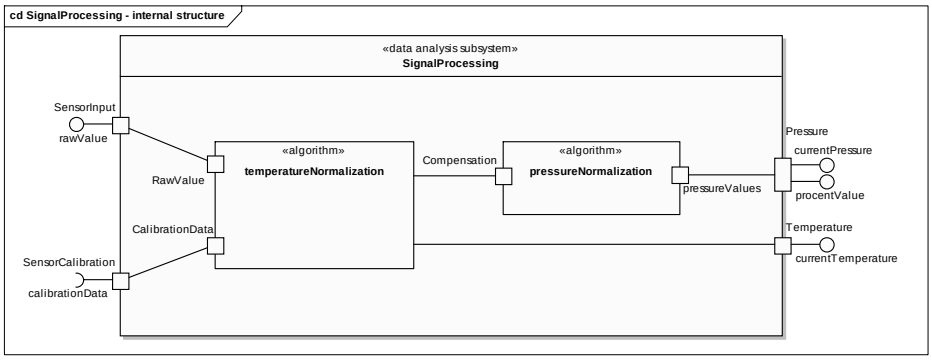

Abbildung 4: Kompositionsstrukturdiagramm des `SignalProcessing` Subsystems

Zur Veranschaulichung stellt Abbildung 4 die interne Struktur des `SignalProcessing` Subsystems in Form eines Kompositionsstrukturdiagramms dar. Es realisiert die Signalverarbeitung eines Messumformers, das heißt seine Hauptfunktionalität besteht darin, Sensor-Rohdaten in physikalische Druck- bzw. Temperatur-Messwerte umzuwandeln, die dann durch das `CurrentOut` Subsystem in ein analoges Ausgangssignal weiter verarbeitet werden. Wie dargestellt, wird diese Funktionalität intern durch zwei Bestandteile (Parts) realisiert, eine Temperatur- und eine Druck-Normalisierung. Wie bereits im vorhergehenden Abschnitt beschrieben, werden die Kontextabhängigkeiten des Subsystems wieder in Form von Ports und Exposed Interfaces beschrieben. Im Gegensatz zum vorherigen Beispiel ist hier aber durch die explizit modellierte interne Dekomposition sichtbar, von welchen internen Bestandteilen des Subsystems die angebotenen Schnittstellen erbracht bzw. die benötigten in Anspruch genommen werden.

Wie man in Abbildung 4 erkennen kann, ist der Grad der Formalisierung, mit der eine interne Dekomposition modelliert wird, frei wählbar. So können die Bestandteile eines Subsystems ausschließlich in Form vollständig gekapselter Parts modelliert werden, die über ihre Ports mit anderen Parts oder mittels Konnektoren mit dem umgebenden Subsystem verbunden werden. Es ist jedoch auch möglich, Parts ohne Ports zu modellieren, die dann direkt mit anderen Parts oder dem übergeordneten Subsystem verbunden werden. Weil die UML2 hier sehr flexibel ist, kann die Dekomposition eines Subsystems so formal exakt beschrieben werden, wie nötig. Insbesondere ist es möglich, dort, wo eine weniger formale Notation eingesetzt werden kann, auf unnötige Formalismen zu verzichten.

4.3 Modellieren auf dem richtigen Abstraktionsniveau

Wie bereits beschrieben, sind die mit den Kompositionsstrukturdiagrammen neu eingeführten Modellierungselemente nicht allein auf den Einsatz in Kompositionsstrukturdiagrammen beschränkt, sondern lassen sich auch in anderen Strukturdiagrammen der UML2 anwenden. So ist es zum Beispiel möglich, in Klassendiagrammen auch Klassen mit interner Struktur zu modellieren, d.h. mit interner Dekomposition in Form von Parts, Ports und Konnektoren. Umgekehrt kann man in Kompositionsstrukturdiagrammen aber zum Beispiel auf eine explizite interne Dekomposition verzichten und nur einfache Attribute und Methoden modellieren, wie sie aus bisherigen Klassendiagrammen bekannt sind, die Kontextabhängigkeiten aber trotzdem mit Hilfe von Ports und required und provided Interfaces formalisieren.

Da wir Subsysteme als die zentralen Bestandteile für Wiederverwendung betrachten, sehen wir die explizite Modellierung der Kontextabhängigkeiten, wie sie in Kompositionsstrukturdiagrammen mit Hilfe von Ports und Exposed Interfaces möglich ist, als unverzichtbar an. Als offen betrachten wir allerdings die Frage, wann es sinnvoll ist, die interne Dekomposition eines Subsystems explizit in Form von Parts und Konnektoren zu modellieren und wann es ausreichend ist, einfache Attribute und Methoden (wie in Klassendiagrammen) anzugeben.

Es ist leicht einzusehen, dass eine generelle Antwort auf diese Frage nicht gegeben werden kann. Basierend auf den in unserer Studie gewonnenen Erfahrungen denken wir, dass die Frage, ob die Dekomposition eines Subsystems explizit modelliert werden sollte oder nicht, sich am besten beantworten lässt, wenn man die vom Subsystem aggregierten Bestandteile bzw. die Kandidaten hierfür betrachtet. Handelt es sich um reine Datenobjekte (d.h. Datenkapseln mit zugehörigen Zugriffsfunktionen) oder um reine Funktionalitäten, so deutet das unserer Meinung nach darauf hin, dass das Abstraktionsniveau zu fein gewählt wurde. Ein weiteres Indiz hierfür ist, dass die Konnektoren zwischen den einzelnen Bestandteilen des Subsystems in diesen Fällen oft reine Datenzugriffe der Funktionen auf die Daten darstellen.

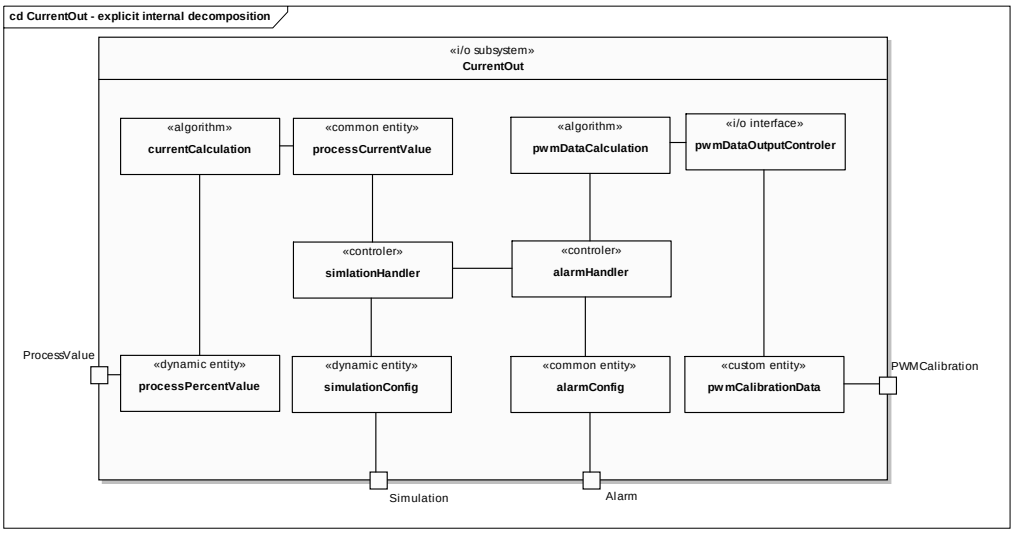

Abbildung 5: Beispiel für ein zu fein gewähltes Abstraktionsniveau

Abbildung 5 veranschaulicht dies am Beispiel des bereits bekannten Subsystems `CurrentOut`. Hierbei handelt es sich bei den mit dem Stereotyp `«entity»` gekennzeichneten Parts im Wesentlichen um reine Datenobjekte, bei den mit `«algorithm»` beziehungsweise `«controler»` stereotypisierten Parts um reine Funktionen. Entsprechend symbolisieren die Konnektoren zwischen den Bestandteilen reine Datenzugriffe.

Wie in Abbildung 5 leicht ersichtlich ist, ist die Folge einer solchen Modellierung auf einem zu feingranularen Abstraktionsniveau, dass selbst einfache Subsysteme wie das beispielhaft aufgeführte `CurrentOut` Subsystem relativ komplex wirken.

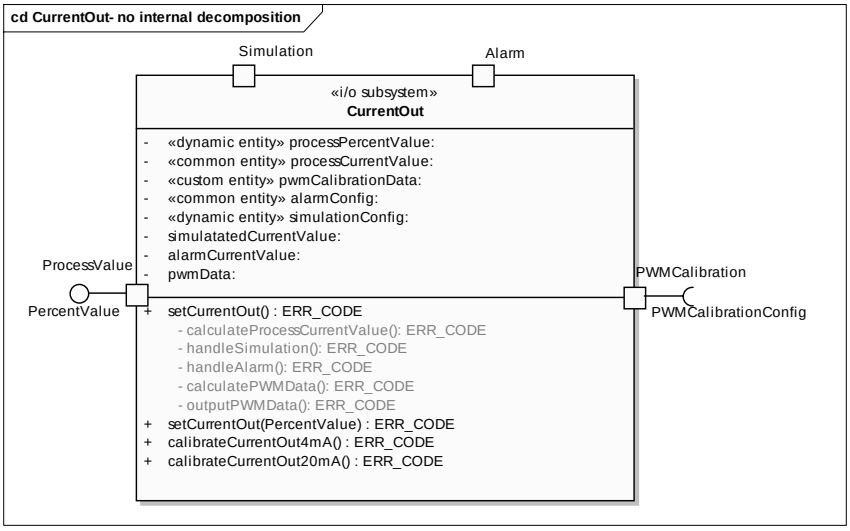

Abbildung 6: Kompositionsstrukturdiagramm für das `CurrentOut` Subsystem (ohne Darstellung der internen Struktur)

Wir betrachten daher eine Modellierung in Form von Attributen und Methoden, wie sie in Abbildung 6 für das Beispiel des Subsystems `CurrentOut` dargestellt ist, in einem solchen Fall als die bessere Alternative.

Da die Frage, ob ein Bestandteil bzw. ein Kandidat hierfür als Part oder einfaches Attribut modelliert werden sollte, im Prinzip für jeden Bestandteil eines Subsystems im Einzelnen geklärt werden sollte und sich nicht für das gesamte Subsystem insgesamt beantworten lässt, betrachten wir es als durchaus sinnvoll, Subsysteme in einer Art Mischform darzustellen, d.h. sowohl mit einfachen Attributen, als auch mit interner Dekomposition. Diese Art der Notation sollte, soweit sich dies aus der UML2 Spezifikation [UML2] der OMG entnehmen lässt, ebenfalls durch Kompositionsdiagramme abgedeckt sein.

Da die von uns betrachtete Software datenintensiv ist (fast jedes Subsystem besitzt eine bestimmte Anzahl von langlebigen Datenobjekten, die sich gut als einfache Attribute des Subsystems modellieren lassen), ist eine solche Mischform der Modellierung von sowohl interner Dekomposition als auch einfachen Attributen in vielen Fällen eine gute Möglichkeit, um die Übersichtlichkeit zu bewahren und unnötige Formalismen dort zu vermeiden, wo sie nicht benötigt werden.

5 Modellieren der System-Architektur

Während auf der Subsystem-Ebene die Kontextabhängigkeiten für jedes Subsystem im Einzelnen modelliert werden, sind auf der Systemebene die strukturellen Abhängigkeiten, die innerhalb des Gesamtsystems zwischen den Subsystemen bestehen, von Interesse. Auch hier besteht die Möglichkeit, ein Kompositionsstrukturdiagramm einzusetzen, bei dem der umgebende Classifier das Gesamtsystem ist und die von ihm aggregierten Parts die zum System gehörenden Subsysteme (streng genommen repräsentieren die Parts dann eine bzw. mehrere Instanzen des jeweiligen Subsystems, sie sind genau genommen durch ein Subsystem typisiert).

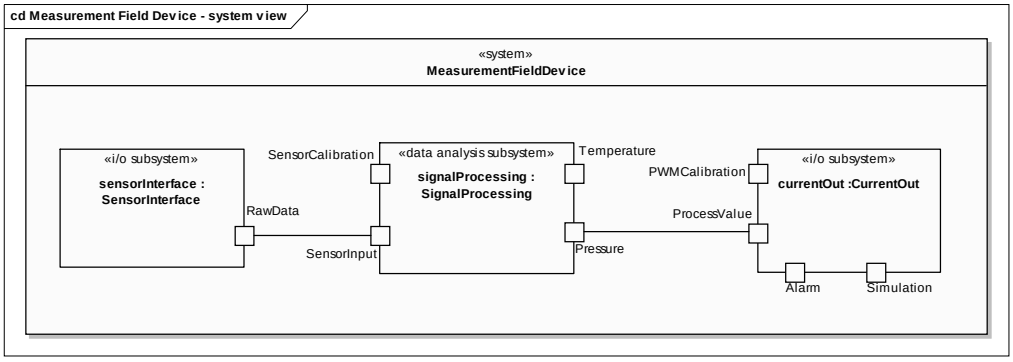

Abbildung 7: Kompositionsstrukturdiagramm für ein (sehr) einfaches Gesamtsystem

Abbildung 7 zeigt dies exemplarisch für ein sehr einfaches Messgerät, bei dem nur die Subsysteme berücksichtigt wurden, die an der in Echtzeit ausgeführten Aufbereitung der Sensor-Rohdaten bis hin zur Ansteuerung des Stromausgangs beteiligt sind. In der Regel ist ein solches Messgerät komplexer und umfasst zusätzlich Subsysteme zur Anbindung an verschiedene Bus-Systeme und zur Ansteuerung von Displays sowie eine Reihe von Subsystemen für Basisdienste wie Speicherverwaltung, Verwaltung von Zugriffsrechten, etc.

Da die im Kompositionsstrukturdiagramm des Systems modellierten Parts durch das jeweils repräsentierte Subsystem typisiert werden, besitzen sie alle vom Subsystem definierten Ports. Hier betrachten wir es, da wir alle Kontextabhängigkeiten der Subsysteme auf der Subsystem-Ebene in Form von Ports und Exposed Interfaces explizit beschreiben, als sinnvoll, nur vollständig gekapselte Parts zu modellieren (d.h. Parts, die Subsysteme repräsentieren, dürfen nur über ihre Ports mit anderen Subsystemen verbunden werden).

Da wir das System zumindest aus architektonischer Sicht als eine in sich abgeschlossene Einheit betrachten, bei dem keine Kontextabhängigkeiten zu anderen Systemen bestehen, besitzt der umgebende Classifier im System-Kompositionsstrukturdiagramm im Gegensatz zu den Kompositionsdiagrammen auf der Subsystem-Ebene keine Ports.

Problematisch ist in diesem Zusammenhang, dass benötigte und angebotene Schnittstellen nur bei den Ports des umgebenden Classifiers in Form von Exposed Interfaces explizit modelliert werden dürfen. Dies ist nicht für die Ports der von ihm aggregierten Parts möglich, denn diese stellen ja nur Referenzen auf die Ports des typisierenden Subsystems dar. Das führt auf der Systemebene dazu, dass die auf Subsystem-Ebene differenziert modellierten Interfaces in der Repräsentation des Subsystems als Part nicht sichtbar sind und strukturelle Abhängigkeiten zwischen Subsystemen lediglich auf der Granularität von Ports modelliert werden können (Konnektoren können auch nur Ports und keine Exposed Interfaces miteinander verbinden). So ist zum Beispiel in Abbildung 7 nicht mehr ersichtlich, welches der beiden Interfaces des Ports `Pressure` des Signalverarbeitungssubsystems mit dem `ProcessValue` Port des `CurrentOut` Subsystems verbunden wird.

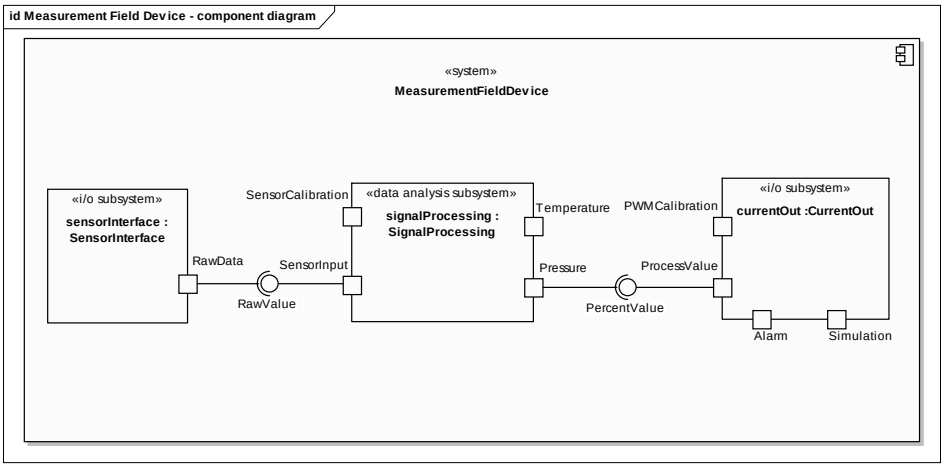

Abbildung 8: Komponentendiagramm für ein (sehr) einfaches Gesamtsystem

Hier wäre möglicherweise eine Notation in Form von *Assembly*-Konnektoren (in der so genannten *Ball*- und *Socket*-Notation) wie sie in UML2 Komponentendiagrammen angeboten wird, besser geeignet. Daher betrachten wir ein Komponentendiagramm, wie es in Abbildung 8 exemplarisch für unser einfaches Messumformer-Feldgerät dargestellt ist, zur Modellierung des Gesamtsystems in den meisten Fällen als die bessere Alternative, wenngleich auch dort, wie in Abbildung 8 ersichtlich ist, die Notationselemente Part und Port von Kompositionsstrukturdiagrammen eingesetzt werden können.

Ein weiteres Problem beim Einsatz eines Kompositionsstrukturdiagramms auf der System-Ebene (das allerdings auch bei Komponentendiagrammen besteht) ist, dass die Darstellung des Gesamtsystems, das im Gegensatz zu oben aufgeführtem Beispiel in der Regel aus einer Vielzahl von Subsystemen besteht, in der Praxis recht komplex und unübersichtlich werden kann, wenn eine Vielzahl von Subsystemen und strukturellen Abhängigkeiten zwischen diesen modelliert werden müssen.

6 Modellieren von Schichten-Architekturen

Bei der von uns betrachteten Domäne der Messumformer-Feldgeräte werden Schichten-Architekturen vielfältig verwendet. So sind insbesondere die Bussysteme, mit deren Hilfe die Feldgeräte mit anderen kombiniert und von außen angesteuert werden können, in Schichten organisiert, was natürlicherweise Auswirkungen auf die Softwarebestandteile hat, die die Busansteuerung übernehmen.

Auch die Architektur eines Feldgerätes selbst besteht üblicherweise aus mehreren Schichten. So gibt es grundlegende Basisdienste, die relativ hardwarenahe Aufgaben, wie die Speicherverwaltung oder die Fehlerdiagnose übernehmen, Subsysteme, die am Echtzeitpfad des Gerätes beteiligt sind (das sind unter anderem alle an der Verarbeitung der Sensor-Rohdaten beteiligten Subsysteme), und solche, die zur Konfiguration des Geräts dienen (wie zum Beispiel Display-Ansteuerung und Bus-Anbindung). Daher stellt sich bei der Modellierung der Architektur auf der System-Ebene die Frage, wie eine Schichtenarchitektur mit Hilfe eines Kompositionsstrukturdiagramms bzw. eines Komponentendiagramms sinnvoll modelliert werden kann.

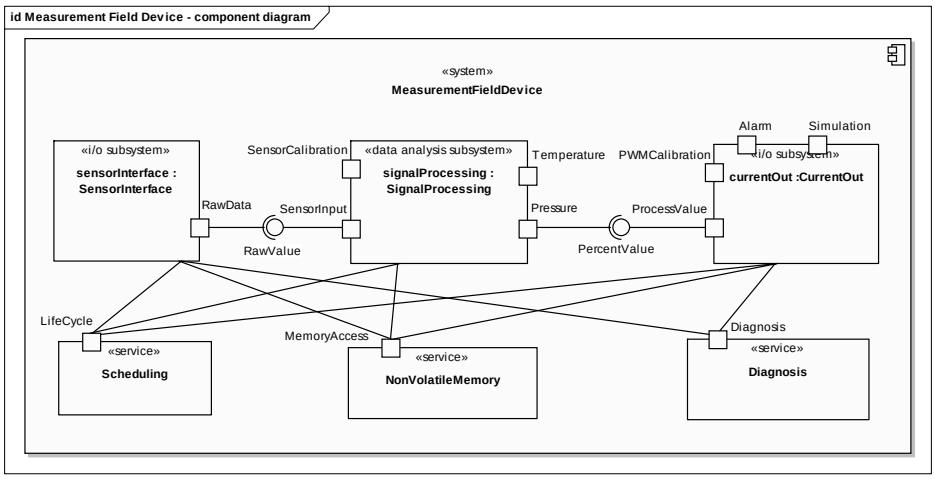

Abbildung 9: Kompositionsstrukturdiagramm einer 2-Schichten-Architektur

Da ein Kompositionsstrukturdiagramm streng genommen nur einen einzigen Classifier und seine interne Struktur darstellt, lässt sich mit ihm auf der System-Ebene eine Schichtenarchitektur praktisch nur, wie in Abbildung 9 dargestellt, durch die Anordnung der Parts (die ja die Subsysteme darstellen) bzw. durch deren Darstellungsart (z.B. durch unterschiedliche Farbgebung) andeuten, da ein explizites Modellierungselement für Schichten in Kompositionsstrukturdiagrammen nicht angeboten wird. Ähnliches gilt auf der Subsystem-Ebene, wenn ein Subsystem selbst aus mehreren Schichten besteht, um beispielsweise von der Hardwareumgebung zu abstrahieren. Dies ist problematisch, da keine präzise Semantik angegeben werden kann, wenn Schichten nicht explizit modelliert werden können (insbesondere sind Kommunikationsbeziehungen zwischen unterschiedlichen Schichten semantisch nicht von denen innerhalb einer Schicht zu unterscheiden).

Ein Lösungsweg besteht darin, für jede Schicht ein eigenes Kompositionsstrukturdiagramm einzuführen (d.h. Schichten als Classifier zu betrachten), in dem die zur Schicht gehörenden Subsysteme in Form von Parts modelliert werden. Das übergeordnete Kompositionsstrukturdiagramm der System-Ebene (bzw. der Subsystem-Ebene im Falle eines mehrschichtigen Subsystems) stellt dann nur noch die Schichten (in Form von Parts) und die Kommunikationsbeziehungen zwischen den Schichten dar. Dies hat den Vorteil, dass in den Kompositionsstrukturdiagrammen der einzelnen Schichten die innerhalb der Schicht existierenden Kommunikationsbeziehungen (ausgedrückt durch Konnektoren zwischen den Parts) klar von denen zu Subsystemen anderer Schichten abgrenzbar sind (ausgedrückt durch Konnektoren zwischen den Parts und den Ports der Schicht als umgebenden Classifier).

Ein Nachteil, der auch dieser Lösung anhaftet, ist, dass die Kompositionsstrukturdiagramme der einzelnen Schichten relativ komplex und schwer überschaubar werden, wenn zum Beispiel – wie im Fall der von uns modellierten Feldgeräte-Architekturen – eine Reihe von Basisdiensten von nahezu allen anderen Subsystemen verwendet werden (die in Abbildung 9 ersichtlichen zahlreichen Kommunikationsbeziehungen zu den Basisdiensten `Scheduling`, `NonVolatileMemory` und `Diagnosis` wären dann in die Kompositionsstruktur-diagramme der Schichten verlagert).

Ein von Selic vorgeschlagener Ansatz zur Modellierung von Schichten-Architekturen besteht darin, die Zugriffe von Subsystemen auf Dienstleistungen darunter liegender Schichten unmittelbar in den Subsystemen selbst, in Form von privaten (also nach außen nicht sichtbaren) Ports zu modellieren (siehe [S03] für Details). Private Ports stellen keine vom System angebotenen Funktionalitäten dar, sondern werden nur zur internen Implementierung verwendet (sie werden nicht auf dem „Rand“ sondern innerhalb des umgebenden Classifiers notiert, wodurch sie von anderen Ports des Subsystems auch graphisch unterschieden werden können). Diese Lösung geht auf den ROOM-Ansatz [SGW94] zurück, dessen Actor-Diagramme als Vorläufer der in UML2 eingeführten Kompositionsstrukturdiagramme betrachtet werden können und lässt sich daher einfach auf Kompositionsstrukturdiagramme übertragen.

Auch wenn dadurch die System-Ebene übersichtlicher modelliert werden kann, halten wir eine solche Lösung nicht für geeignet, da auf der System-Ebene nicht mehr alle Kontextabhängigkeiten von Subsystemen erkennbar sind, wenn deren private Ports Subsysteme anderer Schichten bzw. Basisdienste referenzieren (und im Hinblick auf die spätere Wiederverwendung handelt es sich bei den zur Implementierung verwendeten Diensten anderer Schichten in der Tat um wichtige Kontextabhängigkeiten). Zudem gibt es nach unserer Kenntnis bislang auch kein UML2 Werkzeug, das die Darstellung privater Ports innerhalb des umgebenden Classifiers unterstützt.

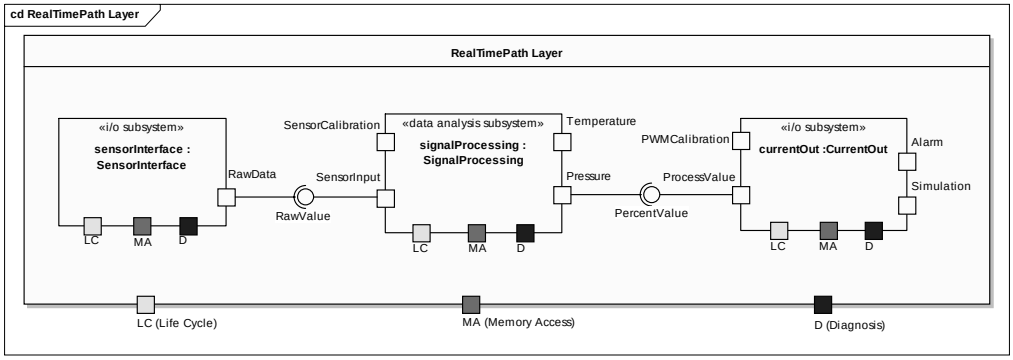

Abbildung 10: Kompositionsstrukturdiagramm der Echtzeit-Schicht des Geräts

Stattdessen schlagen wir vor, wie oben bereits eingeführt, für jede Schicht ein eigenes Kompositionsstrukturdiagramm einzusetzen, hierbei aber (im Normalfall) nur die innerhalb der Schicht existierenden Kommunikationsbeziehungen explizit mit Hilfe von Konnektoren zu modellieren und die Zugriffe auf Dienstleistungen darunter liegender Schichten - wie in Abbildung 10 dargestellt – über die Darstellungsweise der Ports bzw. deren Benennung kenntlich zu machen.

7 Fazit

Die mit Version 2 des UML Standards eingeführten Kompositionsstrukturdiagramme bieten eine Reihe von Modellierungsmöglichkeiten, die so bisher nicht angeboten wurden. Insbesondere durch die Aufhebung der starren Grenzen zwischen den Strukturdiagrammen (die Modellierungselemente eines Kompositionsstrukturdiagramms lassen sich ja auch in anderen Strukturdiagrammen einsetzen) hat die UML vor allem im Bereich der Architekturmodellierung an Ausdruckskraft gewonnen.

So konnten wir bei der von uns betrachteten Domäne die Laufzeitstruktur der Systeme in Form von Parts und Konnektoren modellieren (im Gegensatz zu der rein statischen Struktur, wie sie beispielsweise mit Klassendiagrammen modelliert wird). Auch war es bisher nicht möglich, Kontextabhängigkeiten eines Subsystems mit Hilfe der UML explizit zu spezifizieren. Dies wird nun durch das Konzept der Ports und hervorgehobenen Interfaces deutlich verbessert.

Während Kompositionsdiagramme auf der Subsystem-Ebene gut einsetzbar sind, haben sie sich als weniger geeignet erwiesen, wenn es darum geht, eine Gesamtsystem-Architektur zu modellieren, also das Zusammenspiel einzelner Subsysteme und deren strukturelle Abhängigkeiten zu beschreiben. Hier zeigten sich vor allem Schwächen, wenn es darum geht, Schichten-Architekturen abzubilden.

Was die Unterstützung durch Werkzeuge angeht, so gibt es bislang nur wenige Werkzeuge, die in Anspruch nehmen, UML2 konform zu sein, und von diesen beherrscht – soweit wir das überblicken können – derzeit keines die Modellierung von Kompositionsstrukturdiagrammen exakt so, wie es der UML2 Standard festlegt (zum Beispiel die Darstellung privater Ports). Darüber hinaus haben wir wieder einmal festgestellt, dass der Nutzen der Diagramme stark davon abhängig ist, wie gut das jeweils eingesetzte Werkzeug es erlaubt, die Diagramme als Sichten eines konsistenten Gesamtmodells zu handhaben. So sollte ein Werkzeug zum Beispiel das Referenzieren eines gemeinsamen Modellelementes aus verschiedenen Diagrammen unterstützen, beispielsweise beim Einsatz von Protokoll-Zustandsautomaten zur Beschreibung der Interaktionsmöglichkeiten an den Ports eines Subsystems. Hier können unserer Meinung nach die vorhandenen Werkzeuge noch erheblich verbessert werden.

Trotz der identifizierten Probleme betrachten wir die neu eingeführten UML2 Kompositionsdiagramme und insbesondere die damit auch in anderen Strukturdiagrammen einsetzbaren Modellierungselemente aber als Bereicherung. Wichtig ist wie so oft, sie geeignet einzusetzen. Wenngleich wir die hier geschilderten Erfahrungen in der Domäne von Messumformer-Feldgeräten sammelten, denken wir, dass die Ergebnisse sich auch auf andere Domänen übertragen lassen.

Literatur

[J04] Jeckle, Mario; Rupp, Chris; Hahn, Jürgen; Zengler, Barbara; Queins, Stefan: UML2 glasklar. Carl Hanser Verlag München Wien. 2004.

[NMS04] Nyßen, Alexander; Müller, Peter; Suchotzki, Jan; Lichter, Horst: Erfahrungen bei der systematischen Entwicklung kleiner eingebetteter Systeme mit der COMET-Methode, Proceedings Modellierung 2004, Marburg, März 2004

[S03] Selic, Bran: Modeling Real-Time System Architectures with UML 2.0, Vortrag zur EmSys Summer School, Salzburg, Juni/Juli 2003, http://www.softwareresearch.net/site/other/EmSys03/.

[SGW94] Selic, Bran; Gullekson, Garth; Ward, Paul T.: Real-Time Object-Oriented Modeling, Wiley Professional Computing, 1994.

[UML2] OMG: UML 2 Superstructure Final Adopted Specification, ptc/03-08-02, August 2003.

---

[1] RWTH Aachen, Lehr- und Forschungsgebiet Informatik III,
Ahornstraße 55, D-52074 Aachen, E-Mail: {any|lichter}@informatik.rwth-aachen.de

[2] ABB Corporate Research, Wallstadter Straße 59, D-68526 Ladenburg,
E-Mail: {peter.o.mueller|jan.suchotzki}@de.abb.com

[3] ABB Automation Products GmbH, Schillerstraße 72, D-32425 Minden,
E-Mail: andreas.stelter@de.abb.com

An Extended Perspective on Environmental Embedding

Michael Cebulla
Technische Universität Berlin, Fakultät für Elektrotechnik und Informatik,
Institut für Softwaretechnik und theoretische Informatik,
e-mail:mce@cs.tu-berlin.de

Abstract: We propose a notion of extended embedding which allows to consider multiple aspects of systemic embedding. We claim that a high degree of *adaptivity* is required from advanced systems which reach a rapidly increasing degree of distribution in many fields of application (medical processes, home care, e-business). In many cases mobile applications have to provide safety-critical services in dynamic contexts. Consequently systems have to be able to *act* reliably in unknown or even adverse environments. As a consequence the new requirement of *context awareness* emerges. Consequently there is a need for introducing high level concepts in system specifications. In order to cope with complex situations and dynamic environments systems have to *know* the concepts which are used to describe these contexts by system designers. We propose a method which allows the integrated treatment of multiple environmental aspects. System behavior is conceived as *coordination process* of different systemic components and contextual aspects. Since the underlying semantic models can be mapped to current programming paradigms the environmental models can be integrated into global system architectures.

1 Introduction

At present, the industrial societies are characterized by a technological impulse which was generated by the advances on the field of hardware miniaturization. Small devices (like mobile phones) are getting more and more common reaching a degree of distribution which opens the door for new applications (e.g. in the medical area). Unfortunately new design challenges are coupled with this high degree of mobility. Applications have to offer an increased robustness and a degree of reliability which is unknown up to now in areas concerned (e.g. web based applications). They have to show a reasonable behavior in adverse (or unknown) environments, have to cope with unexpected and undesired events and must tolerate sudden changes in their environment. We claim that these types of scenarios can be considered as generalizations of *environmental embeddings.* Consequency, we propose a notion of *extended embedding* which allows to consider multiple aspects of systemic embedding. Important examples for these aspects are distributed workflows, human factors and environmental conditions.

What is required from this type of systems is a high degree of *adaptivity.* In order to *recognize* certain types of critical situations systems have to *know* the characteristics of these situations and have to be able to choose an appropriate strategy for compensation. As a

consequence of these new requirements of *context awareness* we claim that there is a need for introducing high level concepts in system specifications. In order to cope with complex situations and dynamic environments systems have to *understand* the concepts which are used to describe these contexts by system designers. For example safety critical systems have to know if they are acting in a safe environment. In the light of this complex embedding low level specifications can be considered insufficient. The degree of granularity of environmental modeling has to be situated on the *knowledge level* [Ne82].

One example for these context dependencies is the new impact that human factors have in these kinds of systems. Future ubiquitous services in combination with multimodal interfaces will result in dependencies between user and application whose complexity is unknown up to now. Especially they will have to be able to make reasonable assumptions concerning the identity of users, their goals and their knowledge. Again, we claim that new types of abstractions have to be used for the description of these issues [PCDG02].

On the background of these considerations we propose a method which relies on ontological concepts. One advantage of this method is its support of interdisciplinary systems analysis. It allows for the direct integration of domain specific concepts into (semi-)formal models. By integrating these models into distributed systems we can obtain an internal representation of relevant knowledge in these systems. Thus embedded environmental models enable the systems to show a higher adaptivity in their behavior. This capability is often referred to as *context awareness.*

We first discuss the contribution of our approach to a safety-related engineering of complex systems (Section 2). We then give an account of the use of aspects in our system model (Section 3). In Section 4 - 6 we give examples for the ontological description of systemic aspects. Finally, we show how to deal with complex interactions w.r.t. human error (Section 7) and complex organizational protocols (Section 8). In Section 9 we draw some conclusions.

2 Coordination, Complexity, and Safety

In this paper we adopt a high-level perspective on distributed safety-critical processes in general. We analyse the mechanisms that control the coordination in these systems in order to get a deeper understanding w.r.t. the characteristics and risks of these processes. Generally, one can observe a high degree of robustness of processes which is attained by coordinative adaption to unforeseen environmental situations.

We take our examples from the interdisciplinary analysis of distributed safety-critical processes in sociotechnical systems. Complex sociotechnical systems have evolved to control high risk technologies by teams of highly qualified specialists. These systems can be defined as *complex* safety-critical systems where *teams* of human operators cooperate with *ensembles* of technical units and devices. Usually, the resulting processes are significantly more complex than in systems consisting solely of technological components because they have to be context aware to a high degree. Examples for this kind of systems are atomic power plants, medical operation theaters (where our examples are taken from) and air

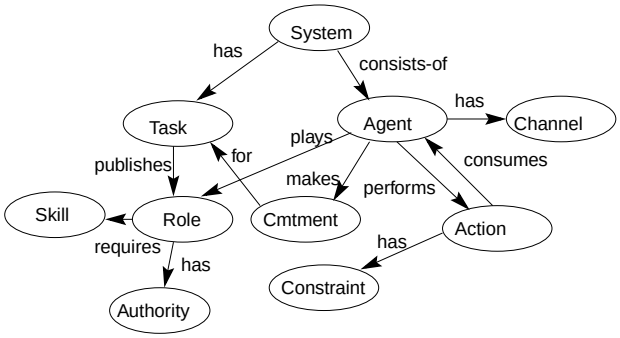

Figure 1: Organizational Ontology (modified from [FBGL95])

traffic control. The need for further model-bases systems analysis is documented by the sad history of catastrophes from Three Miles Island (1979) to Überlingen (2002). The analysis of such complex systems has proven too multi-faceted for the traditional single-disciplinary approach.

A model-based interdisciplinary system analysis is a promising strategy against what Leveson calls *intellectual unmanageability* of high risk systems [Le95]. The increasing complexity and tight coupling in contemporary high risk systems make a safe and efficient management difficult if not impossible. The main source of failure in complex systems is not human error or an erroneous component, but the *complex interactions* between components which is not understood to a sufficient degree [Pe84].

In addition we argue that the results of our research and high-level concepts for the various aspects of contextual modeling can be counted as a contribution to an integrated design of very complex systems (i.e. pervasive context-aware services). We use the chemical metaphor [BB92] for the formal description and analysis of coordinative processes (making heavy use of multiset-based behavioral description). As a result we obtain the specification of a virtual machine which is able to simulate the complex interactions in safety-critical systems.

3 System Aspects

The Target System. Since we are interested in a holistic system analysis we have to start with a top level view on the target system. Following [FBGL95] we start with an organizational ontology to describe the basics of the system. Later we will refine several aspects by additional ontologies.

In Figure 1 we show the overall taxonomy of a complex system. Since we model resources as agents we have two arrows between agent and action. While an action is performed by agents it also consumes agents which represent the resources. Generally the arrows represent 1:n-relations. So a system normally is composed of many agents. Agents may have

many communication links and may perform many activities. Agents may play several roles in order to process systemic tasks. Roles are connected with skills (or capabilities) and authorities. Agents play roles in order to execute tasks. For this sake they perform actions and have to comply with constraints. Agents use communication channels in order to interact with each other.

We describe complex systems as arrays of related sub-models which we call aspects. We distinguish the following aspects: Task, Agent, Communication, Organization (e.g. Commitments and Capabilities), Knowledge. We define a complex system as the ensemble of these aspects.

Definition 1 (Dynamic Model) *A complex system $S = \langle T, A, Com, Org, K \rangle$ is defined as a tuple where T represents agent-specific views on the systems task, A is a multiset of systemic* agents instances (or activities)*, Com represents the set of available communication channels, Org the multiset of relevant organizational features and K the multiset of knowledge bases (beliefs).*

As an agent can perform multiple actions at the same time it is possible that it will appear several times in the the multiset A. In these cases many of the aspect-related specifications will be identical (e.g. subjective task representation and knowledge). At this point of our presentation we treat the constitutive systemic aspects as black boxes. We will use the remainder of this paper to describe some details of these aspects.

Behavior. We use transformation rules in order to describe systemic behavior. Thus, we are able to specify the interactions and side effects between the different systemic aspects.

$$\langle \mathbf{out}(\mathbf{msg}, \mathbf{Com_j}).\mathbf{A_i} \oplus A, \mathbf{Com_j} \oplus Com \rangle \quad \rightarrow \langle \mathbf{A_i} \oplus A, (\mathbf{Com_j} \oplus \mathbf{msg}) \oplus Com \rangle$$

We give a simple example for a systemic transformation. Agent A_i uses his standard action out to put message msg into the communication channel Com_j. This action results in a state where the message is contained in the communication channel.

We heavily rely on the operator $\oplus$ for multiset-union which we sometimes also use (by overloading) to express the union of sets. Note that we leave aside the aspects which we consider to be irrelevant for a specific transformation. As a help for the reader we mark the important activities using bold font.

4 Task Specification

The description of the systemic task is one important aspect of the overall system. For this reason we have to provide an ontology for the specification of this aspect. Note that there is no static description of a system functionality but only a high-level specification of systemic goals. The attainment of these goals is a task which the systemic agents have to solve under consideration of environmental conditions. Our example again is the coordinative planning of medical processes.

Task	reanimation ([0sec,10sec][_,_][_,_], collaps-observed)
Intentions	*achieve* stable-circulation
Conditions	*role* Nursery // different guidelines for non-pros

Body	if (patient-age $\geq$ 8 years) then do-alarm
	pulse-control ([_,_][_,_][_,10s])
	indirect-signs-of-circulation/respiration, movements
	if (apnea) then mask-respiration
	if (cardiac-arrest) then cardiac-massage
	if (ventricular-fibrillation) then
	app-of-defibrillator ([_,3min][_,_][_,_] collaps-observed)
	if (asystoly) then recheck-diagnosis
	safeguard-airways

Figure 2: Task *Reanimation* [WVK$^+$01]

Since medical processes are highly variable we have to provide separated representations for systemic goals and the plans applicated to attain them. While the goals of a medical operation are more or less fixed the sequences of actions (represented by plans) can vary considerably w.r.t. the given situation or subjective preferences. Plans are developed and refined during the processing of the task.

Following [SMJ98] we conceive a task as consisting of these components:

Definition 2 (Task) *We define a task as a tuple* $T = \langle name,$ *time-annotation,* $intentions, conditions, effects, body\rangle$ *which are described below.*

- The symbolic *name* of a task which can be mentioned in descriptions of other (complex) tasks. In these cases the defined task is a subtask of the other task. Names of elementary tasks are similar to uninterpreted action names in behavioral specification.
- Time annotation. We can specify begin (earliest/latest), end (earliest/latest), and duration (minimal/maximal) of a task.
- Intentions (or goals) are high level specifications of system states which have to be achieved, maintained or avoided during or after execution of the plan.
- Conditions are control-mechanisms for executing a plan or its sub-plans. The conditions are used to define this behavior. The availability of resources and agents (with suitable roles) are special cases of activation conditions.
- The body of a plan specifies the subgoals or subplans which have to be achieved to attain the overall task goal.

In Figure 2 we give an example for a task specification using an ontological description. A subset for the international guidelines for cardiopulmonary reanimation is described using a tabular notation. Besides the name of the task we give in the top of the table a very free time related specification which determines that the beginning of the task's execution

should by as near as possible to the event *collapse-observed.* The task's goal consists in the achievement of a stable blood circulation. The entry in the row *conditions* shows that the task as specified in the body is to be executed by persons who had attended a special training.

5 Agents

Agents are the main units for the articulation of the systems structure. Agents are used to represent operational personnel as well as devices. In our approach we silently assume that every relevant agent is described in the static model. In the dynamic model we deal with *agent instances,* which represent the *activities* which are actually executed in the system.

Agent Instances. Agents contain features and may contain other agents. Agent's features describe the input and output *service access points.*

The material and instruments used by the operators in order to fulfill their task are called *resources.* Since resources may have state and behavior we model them as *agents.*

Definition 3 (Agent Instance) *An agent is defined as a tuple $A = \langle id, I, O, Act \rangle$ where I (resp. O) is the multiset of input (resp. output) properties and Act an expression of the action language.*

Workload. When an agent performs several activities in the mean time he is a multiple member of multiset A. This is the reason why we do not provide an operator for parallel composition in our action language. This is a well known characteristic of multiset-based behavioral modeling. An agent which is a member of the actual configuration but has no concrete assignment is represented as an *idle agent* in the multiset A.

For performing certain actions a systemic agent (e.g. a human actor) has to engage with a part of his own resources (e.g. visual sense, one or two hands, acoustic understanding). Obviously these resources are limited. Consequently, a given agent can only perform a number of certain actions at the same time. We model this relation by managing the state of the agent's properties. In addition, we can compute an agent's workload by analyzing the allocation of his features. We can use this model for the detection of over-assignments.

We use cardinality constraints to express the limited availability of resources. Since in our case a systemic agent may be represented by multiple agent instances (due to parallel assignments) we have to consider all agents which a certain id.

$$\langle A_{Env} \oplus Ag \rangle \wedge \#(Ag \lhd (id = id_0 \wedge lman = used) > 1 \quad \rightarrow$$
$$\langle (\mathbf{A_{Env}} \oplus \mathbf{overassignment}(\mathbf{id_0}, \mathbf{lman})) \oplus A \rangle$$

CommitmentIntubation	Until 6
Body	
Delegated by: top	id: a005

Figure 3: Example: Commitment

In this rule we use the *filter*-operator $\lhd$ for multisets and the cardinality-operator $\#$. We claim that there is an overassignment if the cardinality of the set of activities which make use of an agent's left hand is greater than one. Other types of overassignments are described by similar rules.

6 Organizational Issues

The central concept w.r.t. the description of coordination processes is the assignment of tasks to agents. Therefore we refine the coordinative aspect of our system definition in order to provide the concepts of commitment and capability. The concept of role (which is defined as a bundle of capabilities) is not deepened in this paper. We call this aspect the *organizational aspect.*

Definition 4 (Organizational Aspect) *The organizational aspect of a complex system is defined as a tuple* $Org = \langle Comt, Rol, Cap \rangle$ *where* $Comt$ *contains sets of agent-specific* Commitments*,* Rol *agent-specific sets of* roles *and* Cap *agent specific sets of capabilities.*

Commitments. *Commitment* is the central atomic concept for organizational description. It has strong similarity to the concept of *intention* which was taken over to distributed systems from psychology. Commitment however is a stronger concept since it contains social implications. Thus an agent's commitments can be conceived as a specification of his future behavior. The commitments of an agent are used by other agents in order to reason about their own future actions.

Definition 5 (Commitment) *The set* $Comt_i$ *contains the goals for which the agent* A_i *has committed himself.*

In Figure 3 we show an example commitment for an agent with the id a005. This agent has committed to finish the task of intubation until the system time 5. Note that the commitment's body is empty leaving the choice of behavior to the agent's capabilities. This is what we call *open delegation* [FC01]. Another characteristic feature in this example is that the task was not delegated by anybody to the agent. In addition, we have to consider joint commitments which make it possible to delegate task to groups of agents (beyond the scope of this paper).

Agents use commitments to plan the allocation of their resources. The sum of all commitments allows to predict the course of the systems future behavior to a certain degree. This uncertainty in the prediction of behavior results from the agents ability to change their commitments given unexpected environmental conditions.

Last not least we specify the conditions which must be given for an agent a to commit for a task t. The agent a has to have (at least) one capability c which is suited for the fulfillment of t. Agent a has to be authorized to perform c and must have enough expertise. The necessary resources provided by the agent have to be available. In the case of a human operator these resources are for example his hands, his visual and acoustic sense etc. For the time which is scheduled for the execution of task t this resources must not be allocated for the fulfillment of another task. In addition there has to be a free time interval (a *slot*) for the commitment in the agent's plan (cf. Section 8).

Capabilities. An agent's capability attests that it is able to attain a certain goal by the execution of one or more actions (also called plans). The goal may be attainable by alternative actions (or other courses of actions) which the agent may or may not know. Thus the representation of a capability is very similar to a decomposition rule or the definition of a complex task. Sometimes we say that an agent has the capability to *decompose* a task into its subtasks.

The capabilities of agents may be available to different degrees (dependent on his expertise or training). In addition the agent may be authorized to different degrees with respect to the execution of his capabilities.

Definition 6 (Capability) *A capability is an action which can be performed by an agent. A capability may be available to different degrees and the agent's authority to perform this action may be restricted.*

Important examples for capabilities are:

Primitive Actions: The set of actions an agent is able to perform (annotated with the individual degree of availability and authority). Special cases of these actions are allocation and deallocation of resources.

Complex Actions: Agents are capable of performing complex actions which are composed from multiple actions. The definitions of complex actions are used as decomposition rules.

In Figure 4 we show three examples for capability specifications. Each of these capabilities is directed toward the task of intubation. The first two capabilities which are attributed to an anesthesist differ in the course of action. Actually, the first is the normal procedure while the second version describes a variant (called *rapid sequence intubation*) which is applied in emergency situations. The third capability (also directed at intubation) is attributed to an unauthorized person (e.g. a nurse) which has some expertise but is not allowed to perform this activity. If she performed this procedure she would take more time than the anesthesist.

Capability Intubation	Condition normal	id: a005
Body laryngoscopy, intro-tube, check		
Expertise: high	Authorization: high	Duration: 2

Capability Intubation	Condition emergency	id: a005
Body cricoid-pressure, intro-tube, check		
Expertise: high	Authorization: high	Duration: 1

Capability Intubation	Condition normal	id: a008
Body laryngoscopy, intro-tube, check		
Expertise: low	Authorization: low	Duration: 3

Figure 4: Examples for Capabilities

From a global perspective the systemic availability of a certain capability given a certain system state is a critical feature of a specific situation.

A given agent may have also a set of *inofficial qualifications*. These are procedures which are not contained in organizational definitions and may or may not lead to an organizational goal. In many cases these private procedures are problematic or dangerous but in some cases they can be helpful.

Generally the application of certain capabilities is preferred to other capabilities by the organization. The same is true for an individual agent: he has his own preferences concerning the application of capabilities.

7 Human Error

We can integrate current models of human errors in our global system model using our coordination based approach [Re90]. Especially, it is possible to demonstrate that human errors usually are results of complex interactions.

One important type of human error is a *slip*. A slip is an unintended error. We describe slips using the following rule.

$$\langle \mathbf{a}.\mathbf{A_i} \oplus \mathbf{A_{Env}} \oplus Ag, \mathbf{a} \notin \mathbf{Cmt_i} \oplus Cmt \rangle \quad \rightarrow \quad \langle \mathbf{A_i} \oplus (\mathbf{A_{Env}} \oplus \mathbf{slip}(\mathbf{A_i})) \oplus Ag, Cmt \rangle$$

Two special types of *mistakes* can be described by the following rules. In the first rule we describe a knowledge based error: agent A_i believes wrongly that a goal for which he is committed is already attained.

$$\langle \mathbf{A_i} \oplus \mathbf{A_{Env}} \oplus A, (\mathbf{Cmt_i} \oplus \mathbf{a}) \oplus Cmt, (\mathbf{K_i} \oplus \mathbf{done}(\mathbf{a})) \oplus K \rangle \quad \rightarrow$$
$$\langle \mathbf{A_i} \oplus (\mathbf{A_{Env}} \oplus \mathbf{mistake}(\mathbf{A_i})) \oplus A, Cmt, (\mathbf{K_i} \oplus \mathbf{done}(\mathbf{a})) \oplus K \rangle$$

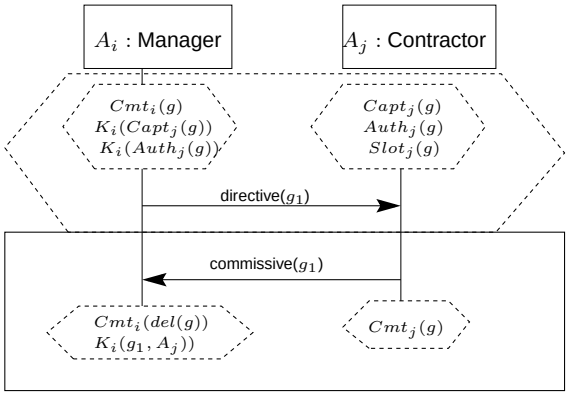

Figure 5: Simple Delegation

The second goal describes a situation where agent A_i employs a capability $r1$ which is not applicable in the given context.

$$\langle \mathbf{A_i} \oplus \mathbf{A_{Env}} \oplus A, (\mathbf{Cmt_i} \oplus (\mathbf{t}|_{\mathbf{r1}}\mathbf{a}, \mathbf{b})) \oplus Cmt, (\mathbf{Cpt_i} \oplus \mathbf{r1}) \oplus Cpt\rangle \wedge \neg applicable(r1) \quad \rightarrow$$
$$\langle \mathbf{A_i} \oplus (\mathbf{A_{Env}} \oplus \mathbf{mistake}(\mathbf{A_i}) \oplus A, Cmt, (\mathbf{Cpt_i} \oplus \mathbf{r1}) \oplus Cpt\rangle$$

8 Complex Interactions

On the basis of the ontological modeling we did w.r.t. important aspects of complex systems and our transition based technique of behavioral description we are able to specify the sophisticated protocols which regulate coordination in complex systems. Especially we can integrate *soft conditions* concerning human factors and organizational features into our description. In our first example we give an exemplary treatment of delegation which is the standard form of distributed decision making in complex systems (as already prepared in Section 6).

Figure 5 shows the basic protocol for the dynamic scheduling of tasks which is common to complex systems [Si94]. At least two agents are involved in a delegation situation where on agents acts as *manager* while the other(s) play the role of *contractor(s)*. For each role we can specify the necessary preconditions (or liveness properties) which have to be given for the beginning of the transaction. So we have to claim that the manager is himself committed for the task he wants to delegate, is convinced that the contractor is capable of achieving the goal and that the contractor is authorized for to fulfill the necessary activities.

In addition we claim that the protocol of delegation is only successful if the directive communication act (described in the diagrams prechart) is followed by a commissive communication act (given the specified side conditions). Thus Figure 5 demonstrates the conse-

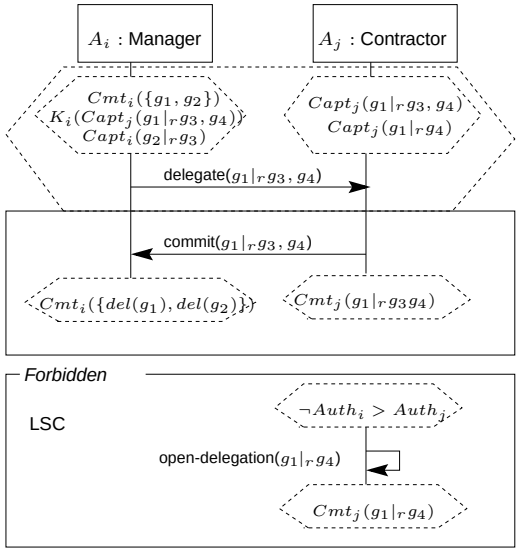

Figure 6: Hidden Delegation/Silent Opening

quences of a successful delegation oriented interaction: now the contractor is committed for the task g which was contained in the managers commitments in the beginning. There it is marked as delegated $del(g)$ after the protocol of delegation is processed.

Finally, in Figure 6 we have the opportunity to demonstrate the safety-critical consequences of some complex interactions which have to follow elaborated protocols. Again we examine a special protocol of delegation. In this case, however, the manager delegates not only the task's goal but he also specifies which activities have to be processed by the contractor to reach this goal. We call this special case of delegation *closed delegation* [FC01]. In our scenario closed delegation is used because the activity g_3 is well-suited to attain another goal g_2. We call this protocol *hidden delegation* because the fulfillment of this second goal is delegated to the contractor without his knowledge.

In our framework we are able to specify an important safety feature for this complex protocol. Consequently in the *forbidden*-part of the diagram we specify that in this case it is forbidden for the contractor to *open* the delegation. The opening of an delegation is a process which can be frequently observed in complex systems. In the context of human interaction for example an actor may change the course of activities which was specified by the manager according to his own preferences (if the manager's authority is not strong enough). Using the visual notation of LSCs we can demonstrate that this may be problematic in certain contexts.

9 Conclusions

In this paper we described a framework for the high-level modeling of complex systems. By choosing an ontological approach for the description of systemic aspects we directly support the integration of domain-specific terminology. Thus we provide a basis for interdisciplinary systems analysis and for simple automated reasoning. Existing tools like Protégé and RACER can be used for this kind of modeling. For the modeling of interactions we can use tools which support the reasoning about MSC-like formalisms (e.g. Harel's Play-Engine). As we already mentioned we can use the chemical metaphor as a semantic basis for the description of coordination processes. Since the semantic concepts involved (e.g. multisets) can be easily mapped to current programming paradigms (e.g. tuple spaces) our high level models can be integrated into reactive system architectures.

References

[BB92] Berry, G. und Boudol, G.: The chemical abstract machine. *Journal of Theoretical Computer Science*. 96(1):217–248. 1992.

[DH01] Damm, W. und Harel, D.: Breathing life into message sequence charts. *Formal Methods in System Design*. 19(1). 2001.

[FBGL95] Fox, M. S., Barbuceanu, M., Gruninger, M., und Lin, J.: An organization ontology for enterprise modelling. *Computers in Industry*. 29:123– 134. 1995.

[FC01] Falcone, R. und Castelfranchi, C.: The human in the loop of a delegated agent: The theory of adjustable social autonomy. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*. 31(5):406–418. september 2001.

[Le95] Leveson, N.: *Safeware. System safety and computers*. Addison Wesley. Reading, Mass. 1995.

[Ne82] Newell, A.: The knowledge level. *Artificial Intelligence*. 18:87–127. 1982.

[PCDG02] Pepper, P., Cebulla, M., Didrich, K., und Grieskamp, W.: From program languages to software languages. *The Journal of Systems and Software*. 60. 2002.

[Pe84] Perrow, C.: *Normal Accidents. Living with High-Risk Technologies*. Basic Books. New York. 1984.

[Re90] Reason, J.: *Human Error*. Cambridge Univ. Pr. 1990.

[Si94] Singh, M. P.: *Multiagent Systems. A Theoretical Framework for Intentions, Know-how, and Communications*. Springer. Berlin, Heidelberg. 1994.

[SMJ98] Shahar, Y., Miksch, S., und Johnson, P.: The asgaard project: a task-specific framework for the application and critiquing of time-oriented clinical guidelines. *Artificial Intelligence in Medicine*. 14:29–51. 1998.

[WVK+01] Wenzel, V., Voelckel, W., Krismer, A., Mayr, V., Strohmenger, H.-U., Baubin, M., H.Wagner-Berger, Stallinger, A., und Lindner, K.: Die neuen internationalen richtlinien zur kardiopulmonalen reanimation. *Der Anaesthesist*. 50:342–357. 2001.

Anforderungen an eine modellbasierte Entwicklung sicherheitskritischer Software für Stellwerkstechnik

Andreas Morawe, Hans-Jürgen Nollau, Murat Şahingöz

Siemens
Transportation Systems
TS RA D E
Ackerstraße 22
D-38126 Braunschweig
Germany

Andreas.Morawe@siemens.com
Hans-Juergen.Nollau@siemens.com
Murat.Sahingoez@TS.siemens.de

Abstract

This article contains a short summary about the process of developing application software for an interlocking system. This interlocking system is intended to be commissioned at German Railways (DB AG). By trying to abstract our experience gained in an ongoing, long-term project, we have started an attempt to define requirements for a model-driven process. In this article, it is not intended to make a contribution to progress in methodology or tool development for model-driven processes. The main concern will be to describe requirements from a more practical point of view.

1 Motivation

Das Entwicklungsteam TS RA D E entwickelt bei Siemens Transportation Systems die Logik für Stellwerke der Deutschen Bahn AG. Im Folgenden soll untersucht werden, ob die bekannten Techniken modellbasierter Entwicklung geeignet sind, die aus sicherungstechnischer Verantwortung und wirtschaftlichen Erfordernissen resultierenden Aufgaben zu lösen.

Im Einzelnen wird auf folgende Themen eingegangen:

- Anforderungen im Projekt,
- Software-Entwicklungsprozess,
- modellbasierter Entwurf (Projektstand und -erfahrungen),
- Anforderungen an modellbasierte Entwicklungsmethoden,
- Arbeitsstand und Vorhaben.

2 Anforderungen im Projekt

2.1 Anforderungen an das Stellwerk

Das von TS RA D E für den Einsatz bei der Deutschen Bahn AG entwickelte Stellwerk ist geeignet, in großen und stark vernetzten Bahnhöfen mit komplexen Betriebs- und Lageplanfällen eingesetzt zu werden. Es hat dabei die höchste

Sicherheitsanforderungsstufe SIL 4 (Safety Integrity Level) nach DIN EN 50128 [1] zu erfüllen. Es muss für höchste betriebliche Anforderungen einschließlich Stadtbahnanwendungen mit Zugfolgezeiten von weniger als zwei Minuten geeignet sein.

2.2 Einordnung der Stellwerkslogik in das Gesamtsystem Stellwerk

Ein modernes Stellwerk besteht aus einem arbeitsteiligen Netzwerk von Rechnern und einer Menge von Prozesselementen in der Außenanlage. Bei der Entwicklung der Stellwerkslogik muss dafür gesorgt werden, dass diese sich in eine vorhandene Infrastruktur aus Produkten von Siemens TS einfügt (Abbildung 1).

Die Stellwerkslogik enthält:

- Funktionen zum Absichern der Signalabhängigkeit (Signalfahrtstellung nur dann, wenn alle befahrenen Elemente in der richtigen Stellung, überwacht und verschlossen sind),
- Funktionen für das Stellen und Überwachen der Außenanlage,
- Funktionen für das Ermitteln der korrekten Signalbegriffe,
- Funktionen für das Rückstellen von Verschlüssen,
- Schnittstellendaten für linienförmige Zugbeeinflussungsanlagen,
- weitere sicherheitskritische Funktionen in einem Stellwerk wie Flankenschutz, nicht grenzzeichenfreie Isolierung und Nahbedienung.

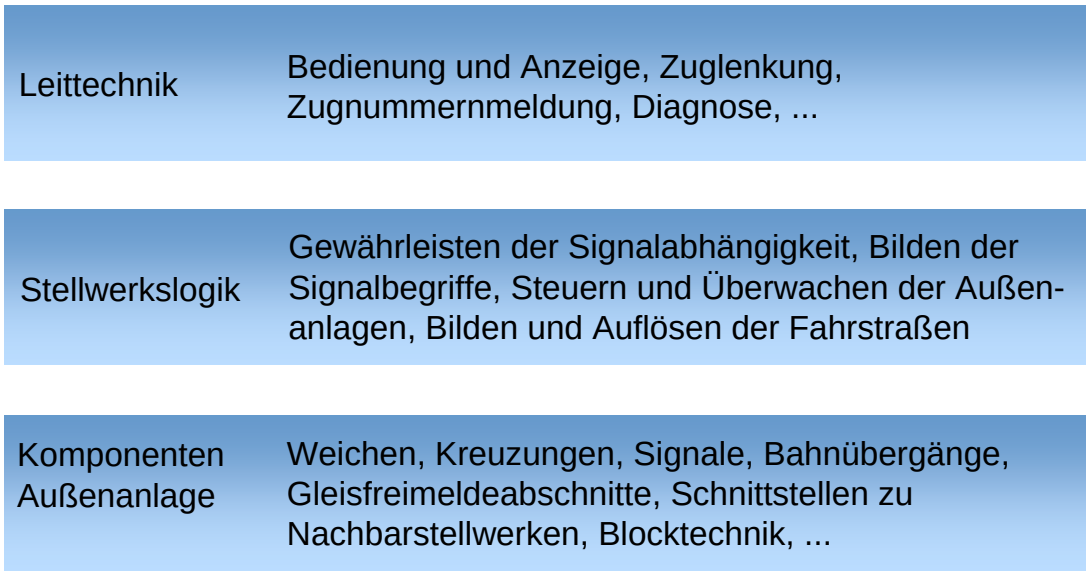

Abbildung 1: Einordnung der Stellwerkslogik in das Gesamtsystem Stellwerk

2.3 Optimierungspotenzial durch Anwenden modellbasierter Methoden

Optimierungspotenzial muss aus zwei wesentlichen Gründen erkannt und genutzt werden:

- Die Stellwerkslogik hat eine Komplexität erreicht, die Hilfsmittel für das Darstellen der funktionalen Wechselwirkungen von Elementen erfordert.
- Die Erfordernisse des Marktes machen es unumgänglich, so schnell wie möglich prototypische Lösungen auf ihre Funktionseigenschaften untersuchen zu können.

Das funktionale Testen der Entwicklungsergebnisse ist sehr aufwändig und bindet erhebliche Ressourcen. Dabei wird anhand konkreter Lageplanfälle nachgewiesen, dass die Anforderungen des Kunden an die Funktionen erfüllt werden.

Hier wären Optimierungen durch das Anwenden modellbasierter Methoden möglich. Das frühzeitige Spiegeln der Kundenanforderungen an einer modellhaften Abstraktion könnte dazu beitragen, Fehlentwicklungen in späteren Entwurfsphasen zu vermeiden.

Sollte es eine Möglichkeit geben, funktionale Anforderungen des Kunden formal zu beschreiben, dann könnten modellbasierte Methoden Schnittstellen für eine formale Verifikation darstellen. Dies könnte die Prüfaufwände reduzieren.

3 Software-Entwicklungsprozess

3.1 Definieren von Elementarten

Im Rahmen der Entwicklung der Stellwerkslogik werden Objekte geschaffen – die so genannten Elementarten. Diese Elementarten repräsentieren sowohl real in der Außenanlage identifizierbare Objekte wie Weichen, Gleisfreimeldeabschnitte und Signale als auch Elemente zum Kapseln von abgeschlossenen Funktionalitäten wie dem Ziel eines Durchrutschwegs oder einer Schnittstelle zu benachbarten Stellwerken (Abbildung 2).

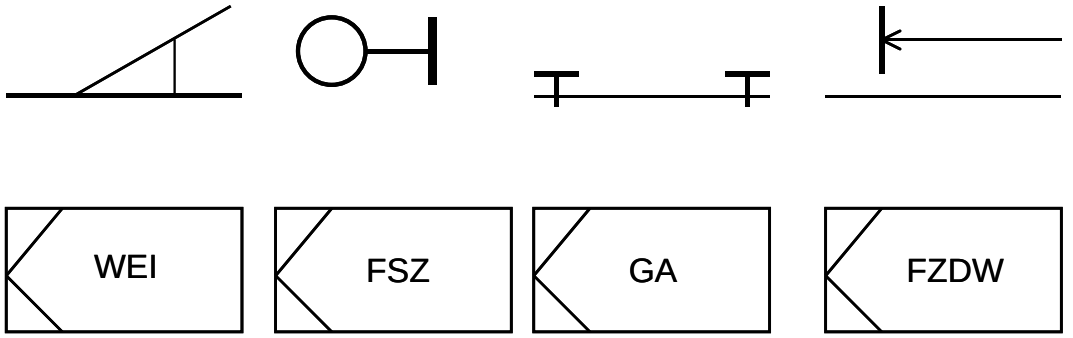

Abbildung 2: Definieren von Elementarten

3.2 Bilden von Objektinstanzen

Die definierten Elementarten kommunizieren miteinander mittels parametrisierter Telegramme in definierten Kommunikationskanälen – so genannten Telegrammkanälen. In den Elementarten werden Telegrammempfangs-

methoden und objektinterne Variablen definiert. Das Verhalten der Elementarten kann durch spezialisierte Variablen – so genannte Projektierungsdaten – konfiguriert werden.

Von den Elementarten werden Objektinstanzen gebildet (Abbildung 3). Diese Objektinstanzen werden über die Telegrammkanäle miteinander verbunden. Im so genannten Elementverbindungsplan wird ein abstraktes Abbild der Bahnhofs- oder Streckentopologie dargestellt. Die Stellwerkslogik stellt somit ein generisches System dar.

Ziel der Entwicklung ist es, eine Typzulassung für die Objekte und die Instanzenbildungsregeln zu erhalten. So können nachfolgend im Seriengeschäft die Anforderungen des Kunden durch einen Projektierungsprozess erfüllt werden.

Die funktionellen Anforderungen an das Stellwerk werden in der Regel nicht für die einzelnen Objekte gestellt (Ausnahme sind die gern als Beispiel verwendeten einfachen Fälle wie Weichenumstellung). Vielmehr gelten die Anforderungen für bestimmte Lageplanfälle, die man als Instanzenbildungsregeln für die Elementarten auffassen kann.

Bestandteil des Entwicklungsprozesses ist ein Zulassungsprozess, der aus der Typzulassung und der Anlagenprüfung besteht. Die Stellwerke werden durch das Eisenbahn-Bundesamt zugelassen.

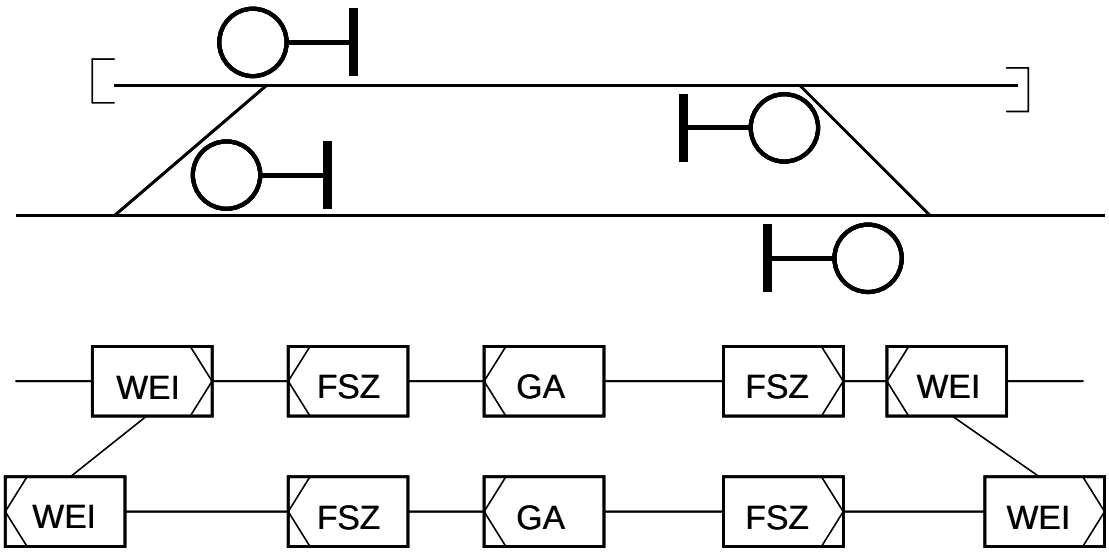

Abbildung 3: Bilden von Objektinstanzen

3.3 Umfang der Steuerungsaufgabe

Für Frankfurt/Main Hauptbahnhof sind ca. 1300 Objektinstanzen allein für das Steuern und Überwachen der Außenanlage erforderlich. Nach den bisherigen Erfahrungswerten ist zusätzlich eine etwa gleich hohe Anzahl Objektinstanzen für logische Funktionen ohne Repräsentation in der Außenanlage erforderlich.

Die Herausforderung beim Entwerfen, Verifizieren und Validieren der Stellwerkslogik besteht weniger in der Anzahl der Objektinstanzen als vielmehr in deren enger Wechselwirkung.

Selbst bei einer sehr trivialen Anordnung von Elementen für eine elementare Fahrstraße (gesicherter Fahrweg zwischen Start und Ziel) sind für den Fahrstraßenaufbau umfangreiche Datenflüsse vorhanden (Abbildung 4).

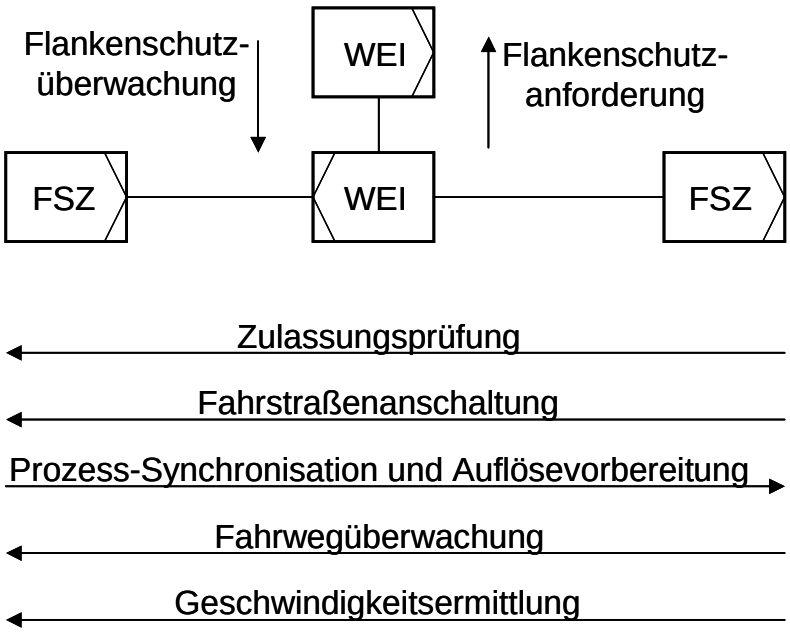

Abbildung 4: Komplexität der Kommunikation für einfache Steuerungsaufgabe

Ein solcher Teilprozess realisiert zum Beispiel:

- das Prüfen auf Zulässigkeit des Fahrwegs (Zulassungsprüfung),
- das Markieren des Fahrwegs (Fahrstraßenanschaltung),
- das Aufbauen und Überwachen des Flankenschutzraums (physischer Schutz einer Zugfahrt vor Gefährdungen aus der Seite/Flanke),
- das zyklische Überwachen des Fahrwegs,
- das Ermitteln des Signalbegriffs.

Praktisch relevante Anwendungsfälle führen schnell zu deutlich komplexeren Informationsflüssen (Abbildung 5). Welche Kommunikationsbeziehungen allein für das Absichern des Flankenschutzes für eine Zugfahrt von einem Beispielsignal aus erforderlich sind, kann in Abbildung 5 nicht mehr übersichtlich visualisiert werden.

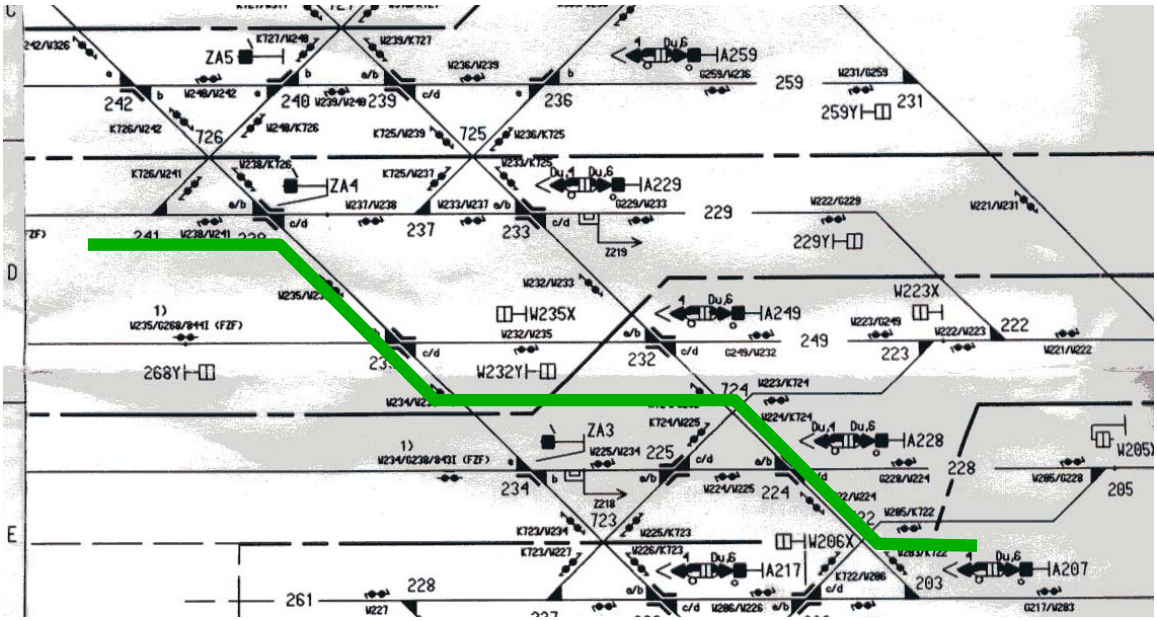

Abbildung 5: Komplexität der Kommunikation für reales Projekt

4 Modellbasierter Entwurf: Projektstand und -erfahrungen

Zum Design der Stellwerkslogik setzt TS RA D E ein Siemens-eigenes domänenspezialisiertes Werkzeug ein: GRACE-FST (Graphical Requirement Analysis and design method in a CENELEC based Engineering process – Functional Specification Tool) [2]. Dieses Werkzeug erlaubt das Definieren:

- von Elementarten, von Telegrammkanälen und Telegrammen einschließlich deren Parameter,
- von elementspezifischen Eigenschaften wie Variablen und Telegrammempfangsmethoden.

Telegrammempfangsmethoden und ihre Unterprogramme und Funktionen werden in einer grafischen Notation editiert. Die Telegrammempfangsmethoden sind weitestgehend lösungsneutral und stellen die Funktionen der Stellwerkslogik dar. Realisierungsnahe Funktionen werden in diesem ersten Arbeitsschritt noch nicht berücksichtigt. Damit soll die dringend notwendige weitestgehende Entkopplung von der Hardware-Plattform erreicht werden.

In einem vom Eisenbahn-Bundesamt zugelassenen Verfahren wird das in GRACE-FST erstellte Design in ein ablauffähiges Programm für das Zielsystem, das reale Stellwerk, transformiert (Abbildung 6). Funktionen wie sicherer Datenaustausch, Ablaufsteuerung, Rechneranlaufbearbeitung, Online-Diagnose, Komponenten für das Absichern des Hardware-Redundanzkonzepts (2v3) werden in diesem zweiten Arbeitsschritt hinzugefügt.

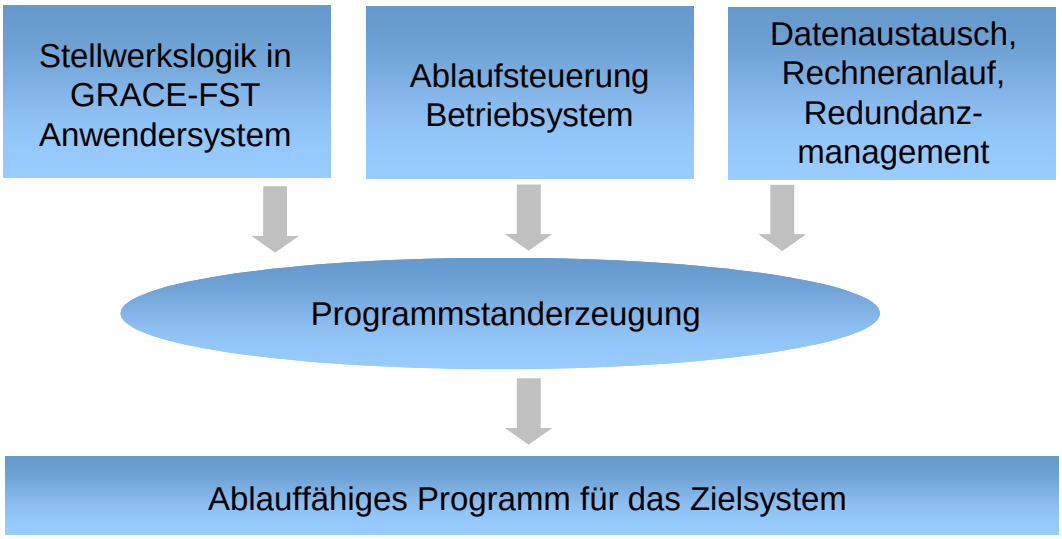

Abbildung 6: Automatisiertes Überführen in das Zielsystem

4.1 Abstraktion der Stellwerksfunktionen

In einem Bottom-Up-Verfahren wird in GRACE-FST ein Low-Level-Design mit Ansätzen zur modellhaften Abstraktion erstellt. Über den Prozess in GRACE-FST hinaus sind zusätzlich abstrahierende Beschreibungsmittel für eine schrittweise Lösungsentwicklung wünschenswert. Dies wird insbesondere für solche Funktionen vermisst, die einen hohen Innovationsgrad im Projekt besitzen. Beispiele für solche Funktionen sind der Flankenschutz und die Regelauflösung (zugbewirkte Grundstellung von Verschlüssen).

Durch das Verwenden der erprobten Methodik SA/RT (Structured Analysis with Realtime Extensions) und des Spezifikationswerkzeugs Statemate® wurden erste Schritte getan, eine solche Abstraktionsebene zu schaffen (Abbildung 7). Zwischen diesen abstrakten Beschreibungsmitteln und GRACE-FST ist allerdings eine manuelle Transformation erforderlich.

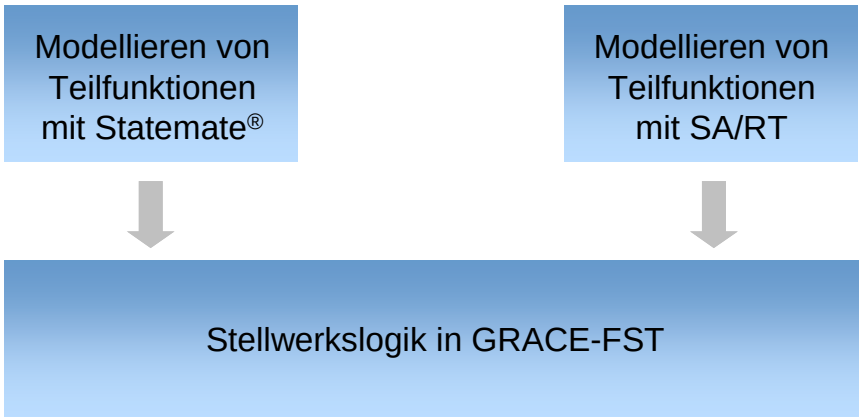

Abbildung 7: Abstraktion der Stellwerksfunktionen

Mit dem Werkzeug Statemate® wurden differenzierte Erfahrungen gesammelt:

- Die Grundidee und die Beschreibungsmittel sind hervorragend zum Beschreiben von komplexen Funktionen geeignet.

- Das Modellieren zwingt zu einem systematischen Vorgehen. Damit wird die systematische Spezifikation auch schwieriger Funktionalitäten gefördert.
- Die Grundidee von frei konfigurierbaren Bedienoberflächen und animierbaren Modellen ist der richtige Ansatz für ein „Rapid Prototyping“.
- Die Kooperation zwischen den kommerziellen Werkzeugen und GRACE-FST entspricht jedoch nicht den Anforderungen. Darüber hinaus erfüllen die gewählten kommerziellen Werkzeuge nicht die Anforderungen an die Stabilität, die Nutzerfreundlichkeit und das Beherrschen komplexer Steuerungsaufgaben in einem generischen System.
- Werkzeugtechnische Beschränkungen und Mängel verhindern das Ausnutzen der prinzipiell leistungsfähigen Beschreibungsmittel für komplexe Funktionalitäten. Das Beschränken auf eine abgeschlossene Teilfunktion ist zwingend erforderlich.

4.2 Einsatzmöglichkeiten modellbasierter Methoden im Projekt

Folgende Optimierungsmöglichkeiten sind durch das Anwenden modellbasierter Methoden zu erkennen:

- Anfertigen verschiedener Sichten auf die Stellwerkslogik (Automat, Sequenzdiagramme, ...),
- Visualisieren der Funktionen verschiedener Lageplanfälle,
- „Rapid Prototyping“ mit iterativer Entscheidungsfindung und Erweiterung bis zur zulassungsfähigen Lösung,
- Generieren von Testfällen,
- Diagnose bei Fehlfunktionen,
- Unterstützen von Verfahren der formalen Verifikation.

4.3 Besondere Anforderungen generischer Systeme

Abstraktionen eines einzelnen Elements sind für die sicherungstechnische Praxis nicht von Relevanz. Stattdessen werden Sichten auf – teilweise sehr komplexe – Lageplanstrukturen erforderlich, die durch definierte Instanzen der Elementarten entstehen. Daraus leitet sich ggf. ein großer Zustandsraum der interagierenden Automaten ab. Methodische und intuitive Möglichkeiten zum Beherrschen dieser Explosion des Zustandsraums sind erforderlich.

4.4 Vision für Werkzeugentwicklungen

Stellt man die modellbasierte Entwicklung in einen weiteren Kontext, ergibt sich die in Abbildung 8 dargestellte Vision für eine Methoden- und Werkzeugentwicklung. Optimierungen lassen sich erreichen, wenn

- die modellbasierte Entwicklung in einen iterativen Entwicklungsprozess eingebunden wird,
- Weiterentwicklungen auf verschiedenen Detailebenen (Prototyping) stattfinden,
- auf die Funktionalität des schon entwickelten Systems zurückgriffen wird.

Dabei sollte es ohne Belang sein, ob Design und Spezifikation auf der Ebene von GRACE-FST oder des Automaten einer Elementart stattfinden. Die Automaten von Elementanordnungen und die Sequenzdiagramme für Elementanordnungen stellen dann nur eine andere Sicht auf die Stellwerkslogik dar. Eine zentrale Rolle nimmt hier das formale Beschreiben der Elementanordnungen ein.

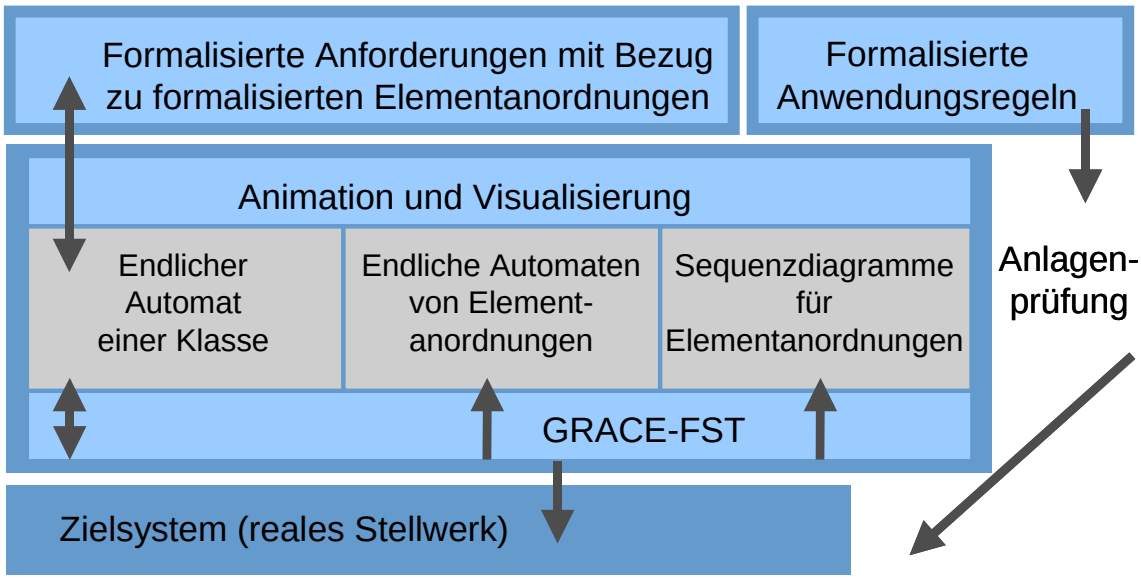

Abbildung 8: Vision für Werkzeugentwicklungen

5 Anforderungen an modellbasierte Entwicklungsmethoden

5.1 Anforderungen aus technischer Sicht

Um Abstraktionstechniken wie Transitionsdiagramme für einen komplexen Wirkungszusammenhang verwenden zu können, müssen aus einer Ansammlung von Variablen semantisch zusammengehörige Zustandsräume gebildet werden.

In gleicher Weise müssen Datenflüsse (Telegramme) ebenfalls abstrahiert werden. Es bedarf noch genauer Voruntersuchungen, um derartige Schritte abwärtskompatibel in das laufende Projekt einordnen zu können.

Langfristig ist es erstrebenswert, die funktionalen Anforderungen des Kunden und die nachfolgende Software-Produktionskette miteinander zu verbinden. Alle Beschreibungsmittel auf diesem Gebiet müssen sich jedoch daran messen lassen, ob sie intuitiv anwendbar sind und im Umfeld generischer Systeme eingesetzt werden können.

Beschreibungsmittel, die nicht gleichzeitig die Aufgabe der Testvektorgenierung und ggf. der Synthese von Testanordnungen lösen, sind als nicht geeignet einzustufen. Ziel ist es, mit optimierten Testverfahren den Aufwand bei der Typprüfung (d. h. der Prüfung für eine Stellwerksbauform) zu verringern und die Testqualität bei der Prüfung konkreter Anlagen beizubehalten oder gar zu verbessern. Die Literatur zeigt einige interessante Ansätze [3, 4], die sich nach

sorgfältiger Prüfung jedoch als noch nicht praxistauglich erwiesen haben [5]. Hier besteht offenbar Handlungsbedarf.

Die Anforderungen aus technischer Sicht lassen sich wie folgt zusammenfassen:

- kein substanzielles Verändern der domänenspezialisierten Werkzeuge (GRACE-FST),
- Abbilden komplexer Systeme in verschiedenen Sichten (Automat, Sequenzdiagramm, ...) mit stabilen, funktionalen Werkzeugen,
- Unterstützen von „Rapid Prototyping“ und iterativer Entwicklung,
- methodische Schnittstellen (temporale Logik, Prädikatenlogik u. ä. sind interessant, aber nicht vermittelbar),
- Unterstützen generischer Systeme (Automatenmodelle, Sequenzdiagramme u. ä. von Lageplanfällen),
- Schnittstellen zu einer methodischen Anforderungsbeschreibungssprache mit der Möglichkeit der Testfallgenerierung.

5.2 Anforderungen aus wirtschaftlicher Sicht

Mit der aus Eigenentwicklung stammenden Werkzeugkette GRACE-FST sind stabile Werkzeuge geschaffen, die auch komplexe Anforderungen umsetzen können. Für Werkzeuge zur Abstraktion wird ein ebenso hohes Niveau erwartet. Ein iteratives Vorgehen hat dabei hohe Erfolgsaussichten. Werkzeuge und Methoden haben in den jeweiligen Iterationsschritten den Nachweis ihrer Tauglichkeit aus technischer und wirtschaftlicher Sicht zu erbringen.

Die Anforderungen aus wirtschaftlicher Sicht lassen sich wie folgt zusammenfassen:

- Stabilität und Intuitivität der Werkzeuge,
- Rückfluss von Investitionen in Methodenentwicklung und Werkzeugentwicklungen/-adaptionen innerhalb der üblichen Entwicklungszeiträume für Weiterentwicklungen der Stellwerkslogik (Erfahrungswert: drei Jahre),
- Kosten für Methodenentwicklung und Werkzeugentwicklungen/-adaptionen in begründbarem Verhältnis zu den übrigen Entwicklungsaufwendungen.

6 Arbeitsstand und Vorhaben

Kernkompetenz von TS RA D E ist die Eisenbahnsicherungstechnik. Alle Überlegungen zu einer modellbasierten Entwicklung sind Werkzeuge und nicht Ziel der Entwicklung selbst.

6.1 Abgeschlossene Aktivitäten

Auf dem Weg zum zuvor dargestellten „Idealprozess“ hat TS RA D E bereits wichtige Ziele erreicht.

So kann die Stellwerkslogik mit dem stabilen Werkzeug GRACE-FST entworfen werden. Das Werkzeug bietet erste Abstraktionsmöglichkeiten.

Der gesamte Prozess vom Design in GRACE-FST bis zum Zielsystem ist automatisiert. Die Konsistenz der auf dem Weg zum Zielsystem erforderlichen Zwischenschritte ist nachweisbar. Der Nachweis ist vom Eisenbahn-Bundesamt anerkannt.

Die Sicht auf die Stellwerkslogik (über eine XML-Schnittstelle) ist für Auswertungswerkzeuge gut geeignet. Sie erlaubt automatisierte Schnittstellen zum Projektierungswerkzeug und das automatisierte Generieren von Anwenderdokumentationen. Die Stellwerkslogik wird mit Werkzeugen ausgewertet. Dabei wird das Nichtvorliegen von bekannten Designfehlern verifiziert, und erste Schritte in Richtung einer Änderungsauswirkungsanalyse werden gemacht.

Der Entwicklungsprozess wird durch Mitarbeiter getragen, die das gesamte notwendige Know-how vom eisenbahnsicherungstechnischen Spezialwissen bis zur Softwareentwurfstechnologie abdecken. Basierend auf langjährigen Erfahrungen besteht Konsens, dass Abstraktionstechniken für das Analysieren und Spezifizieren erforderlich und nützlich sind.

Die Stellwerkslogik kann auf einer leistungsfähigen Testumgebung emuliert werden oder auf dem Zielsystem ablaufen. Testkataloge sind regressionsfähig und sowohl auf dem Zielsystem als auch der Emulation anwendbar. Testziel ist der Nachweis darüber, dass die funktionalen Anforderungen erfüllt sind.

6.2 Laufende Aktivitäten

Werkzeuge zur Verifikation der Stellwerkslogik werden weiter entwickelt. Zielsetzung ist dabei nicht die formale Verifikation der Übereinstimmung von Anforderung und Design, sondern die Freiheit von bekannten Entwurfsfehlern.

Erste Schritte in Richtung Änderungsauswirkungsanalyse sind in der Entwurfsumgebung eingeleitet.

Eine formale Beschreibung der Lageplanstrukturen ist in Arbeit. Eine solche Beschreibung gilt als unverzichtbar, um zukünftig funktionale Anforderungen allgemein gültig formulieren zu können.

Das Änderungsmanagement wird ebenfalls optimiert. Schwerpunkt ist zurzeit das sichere Erkennen und Verfolgen von Änderungen im Design.

6.3 Weiteres Vorgehen

Wenn die zuvor dargestellten Arbeiten einen positiven Effekt gezeigt haben, sind folgende Aktivitäten geplant:

- Schnittstellen zu abstrakteren Beschreibungsmitteln für Lageplanfälle herstellen,
- Funktionsanforderungen mit formalen Ansprüchen definieren, um automatisch Testfälle und Testumgebungen zu generieren,
- Anlagenprüfung rationalisieren.

Die erforderlichen theoretischen Vorleistungen können nicht in einem Entwicklungsprojekt erbracht werden. Hierzu kooperiert Siemens mit Partnern an der TU Dresden, der FH Braunschweig/Wolfenbüttel und der TU Braunschweig.

7 Zusammenfassung

Nach Auffassung der Autoren kann man durch das Anwenden modellbasierter Entwicklungsmethoden einen wirkungsvollen Beitrag dazu leisten, das Spannungsfeld zwischen Qualität und Wirtschaftlichkeit der Software für die konkrete Entwicklungsaufgabe zu beherrschen.

Dazu muss das Entwickeln von Methoden und Werkzeugen auf die Anforderungen komplexer Softwareprojekte in einem industriellen Umfeld ausgerichtet werden. Zur Erfüllung der beschriebenen Anforderungen sind weitere Aktivitäten erforderlich.

Literatur

[1] DIN EN 50128: Bahnanwendungen – Telekommunikationstechnik, Signaltechnik und Datenverarbeitungssysteme – Software für Eisenbahnsteuerungs- und Überwachungssysteme, 2001

[2] B. Jung: Die Methode und Werkzeuge GRACE, Vortrag anlässlich der Tagung FORMS2000, Braunschweig 2000, http://www.ifra.ing.tu-bs.de/forms/aktuell/workshops/2000/forms_2000.html

[3] C. Trog / L. Eriksson: Spezifikation von Stellwerkslogik mit formalen Methoden, SIGNAL+DRAHT, Heft 1+2/2004

[4] A. Rech: Transformation von Stellwerkslogiken in die Eingabesprache eines Model Checkers, TU Braunschweig, 2004

[5] M. Sahingöz: Analyse der Testmethodik für die Betriebsordnung von Stellwerken der Siemens AG, Siemens AG, 2004

μFUP: A Software Development Process for Embedded Systems

Leif Geiger, Jörg Siedhof, Albert Zündorf

University of Kassel, Software Engineering Research Group,
Department of Computer Science and Electrical Engineering,
Wilhelmshöher Allee 73,
34121 Kassel, Germany
{leif.geiger, albert.zuendorf}@uni-kassel.de
http://www.se.eecs.uni-kassel.de/se/

1 Introduction

Large embedded systems like manufacturing halls or complex machines usually employ quite a number of embedded control units. These control units work together either implicitly or explicitly in order to achieve an overall task e.g. manufacturing or transporting some good. In addition, the system of embedded controllers might collaborate with usual personal computers or hosts running e.g. a production control system or a web server or just control panels for production staff.

Designing, implementing and running such a large system with collaborating embedded components is a challenging task due to the complexity of the overall system. Multiple disciplines as e.g. mechanical engineers, electrical engineers and software engineers may be involved. The software of different embedded controllers may be developed by different teams (from different enterprises). Different embedded controllers may utilize different technologies and programming languages, e.g. PLCs, microcontroller programmed in C, FPGAs programmed in VHDL, hosts programmed in Java running a relational database.

Next, the software for the various components is developed while the target manufacturing halls or target machines (and the embedded controllers) are still under construction. From the software engineering point of view, this creates the problem that the software is employed and tested only after the mechanical and electrical components have been build. This creates a lot of time pressure for the software development team in order to get the system in production. In addition, such a setting prevents the application of modern software engineering techniques especially of iterative software development processes where new functionality is build on top of running (and validated) old functionality.

A common approach to enable concurrent development of the different embedded components is the definition of interfaces and communication protocols. The interfaces may be

provided as UML class diagrams specifying which operations are provided by which kind of component. Protocols define valid orders of messages changed between components, this may be specified e.g. with statecharts, cf. [SGW94].

Due to our experiences, interface and protocol definitions are not sufficient in order to enable concurrent development of different embedded components of some complex system. Embedded components development that way frequently do not collaborate immediately but require major adaptions during placing into operation.

In order to improve this situation, we propose to develop an overall simulation of the whole system as some kind of living requirements specification. This overall simulation should provide stubs for all embedded components and it should be possible to validate the collaboration of the components by (automatic) system wide use case tests.

In order to enable the development of a single embedded component prior or parallel to the development of the mechanical and electrical system elements, there should exist a (simple) simulation of all I/O devices of that embedded component. This means, we need an operational model of sensors, actors and busses attached to an embedded system at an appropriate logical level of abstraction. This allows to develop the logical control software of the embedded system in parallel to the hardware development. Combined with an overall system model, this allows to test the embedded component software in an virtual environment reflecting its planned use.

As soon as the hardware for the embedded component and the mechanical components controlled by it are available, the overall simulation may be deployed in order to test the actual embedded component in the context of its later use. Step by step, the components of the overall simulation may be replaced by actual components until the overall system is in production.

An component and overall system simulation may also be used for failure mode effect analysis purposes.

Even for components or for a system in production, the simulation may still be exploited for failure detection and analysis. The actual system and the simulation may be run in parallel and the actual and the simulation behavior may be compared. In case of deviation, either the actual system or the simulation is malfunctioning.

In the next section, we illustrate our ideas with the help of a very simple example. Then we conclude and sum up.

2 Example

This section illustrates our ideas with the help of a simple example, a LEGO carousel storage system controlled by a Java programmable micro controller, cf. Figure 1.

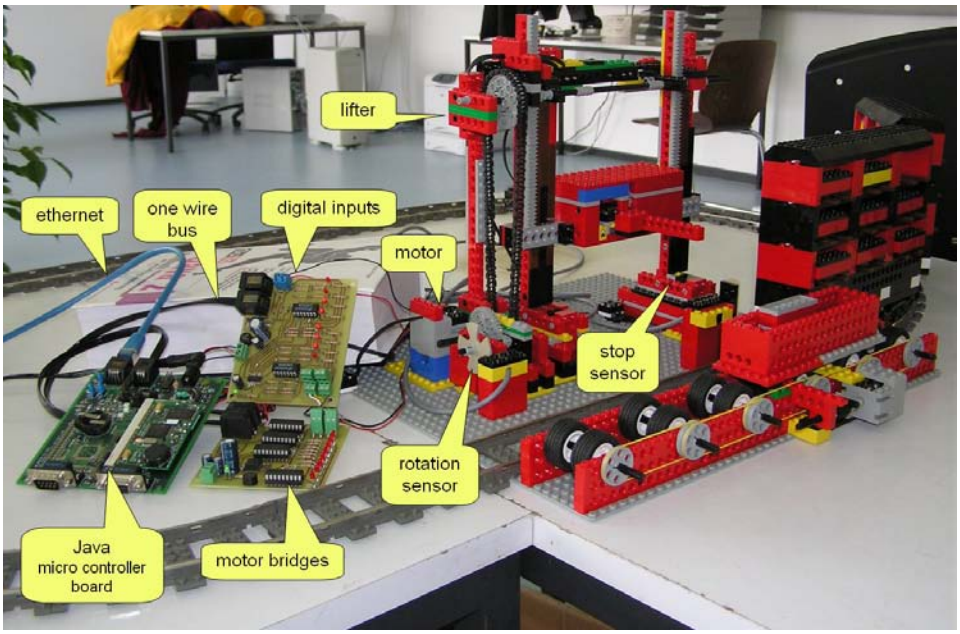

Figure 1: LEGO model of a carousel storage

Following the test-first-principle of modern agile development processes, before we actually program the micro controller, we first set up a test scenario involving simulations of the relevant physical components. Figure 2 shows a coarse grain model of our carousel storage. The carousel consists of a set of racks where each rack consists of certain compartments where each compartment may store some good.

In the scenario of Figure 2, the storage is asked to retrieve some *Aspirin*. In the second activity of Figure 2, the storage looks up where the Aspirin is stored using a qualified goods link. Then the train is asked to move to the position of the corresponding rack. Next the lifter has to raise to the level of the corresponding compartment and finally the pusher arm is asked to push the good our of the rack.

Depending on the complexity of the train, lifter, and pusher component, these components may be controlled by a common micro controller or these components may employ their own micro controller, each.

We propose to refine the behavior of each component on the level of a simulation, first, and to decide about placing of functionality on controllers, afterwards, when the complexity is known.

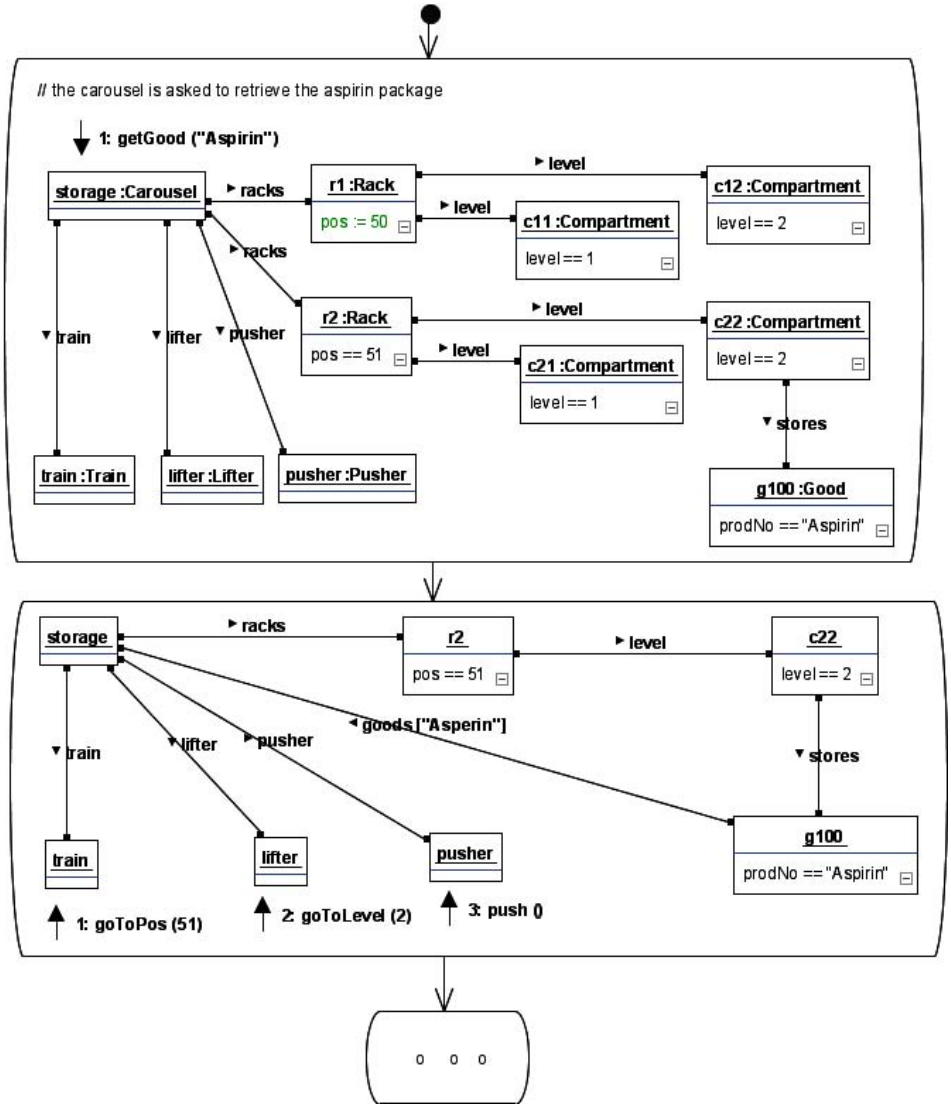

Figure 2: Storage model

Figure 3 shows a simple simulation scenario for the lifter component of our model.

The lifter deals with sensors and actors. Implemented within an embedded controller, the lifter 'sees' its sensors and actors through certain I/O ports, only. In our example, the sensors and actors are connected to a OneWire bus. The motor bridges are addressed via an OneWireOut port and the sensors are addressed via an OneWireIn port, cf. first activity of Figure 3.

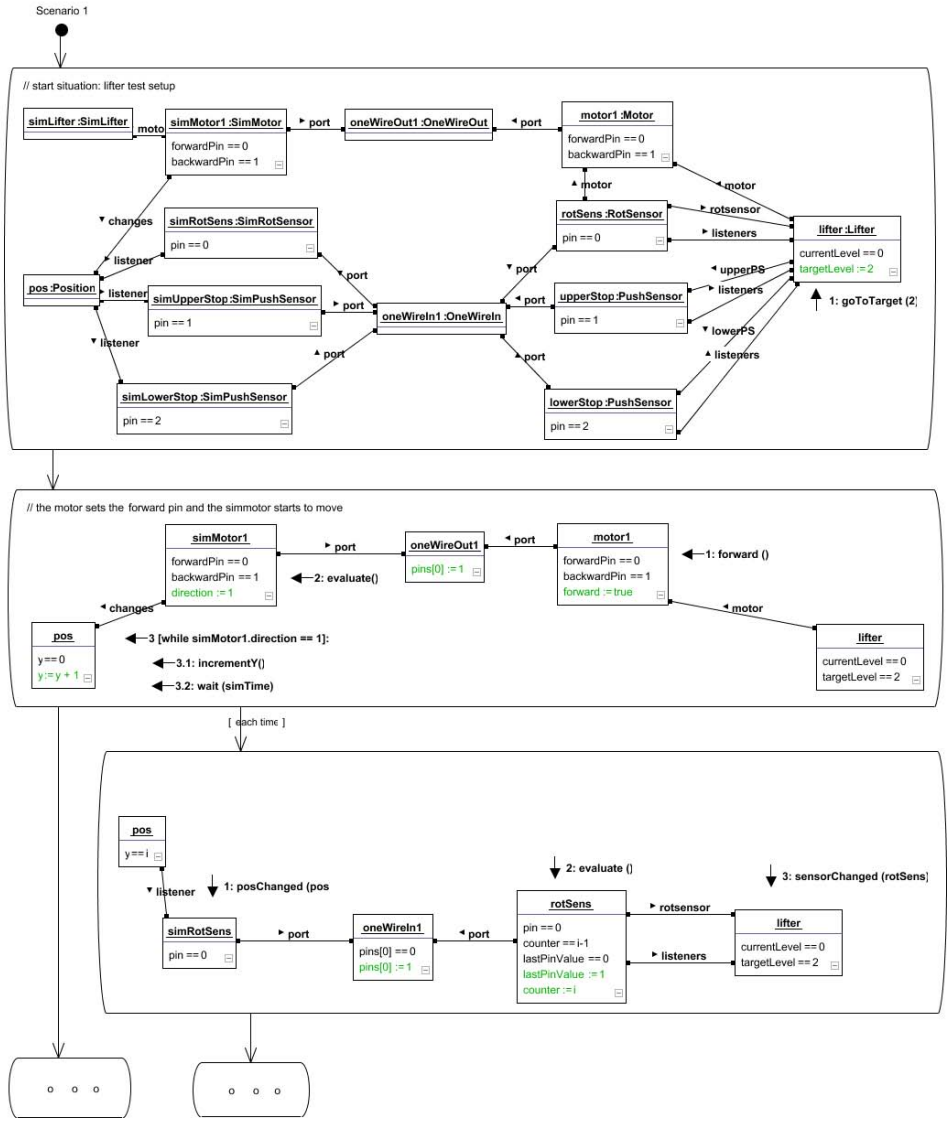

Figure 3: Lifter model

The first activity of Figure 3 shows the OneWire ports in the middle of the object diagram. On the right there are components of the actual control software going to run on some Java micro controller. These control components get and send their signals from and through the OneWire ports. On the left of the object diagram simulation components are shown. These simulation components listen to the OneWire ports.

In the first activity of Figure 3 the lifter is asked to raise to level 2, see method call on

the right. The lifter determines that it has to move upwards and therefore the lifter asks its motor to move forward. The motor sets the corresponding pin in the OneWireOut port. Now the control software has initiated the raising and it waits for signals from the rotation sensor.

The simulation motor listens to its OneWireOut port and thus it is informed that the control software wants the motor to turn forward, cf. step 2 of Figure 3. To simulate the motor movement, the simulation motor periodically updates the y position of the lifter.

In the third step of Figure 3, each incrementation of the y position fires a posChanged event in the listening sensors. The sensors, in our case the rotation sensor analyse the position change and change the values of the corresponding OneWireIn ports, accordingly. This again is recognized by the sensor components on the Java controller. This components believe that actually the lifter raises and that this is observed by the rotation sensor. Accordingly, the rotation sensor living in the micro controller updates its bookkeeping of the lifters positions. This again is observed by the lifter component on the micro controller. This component checks whether the desired height is reached and then it will stop the motor which will reset the corresponding output port which causes the simulation motor to stop signalling position changes.

3 μFUP

The previous section outlines an object oriented modeling of a simple carousel storage. This model employs a number of control components for the lifter, the train, the pusher and for motors and sensors. The model is described at two different levels of granularity: A coarse grain level outlining the collaboration of the top-level carousel components for the retrieval of a certain good, and a fine grained level dealing with sensors and actors of the lifter component. Our model covers the actual control components as well as the physical components. μFUP proposes the following steps in the development of such a system of embedded components:

Developing an overall process simulation platform: Following our approach, one first builds the simulation at the coarse grained level. Building the simulation is conventional software development, thus e.g. the usual Fujaba process (FUP) may be employed, cf. [GSZ03b, DGMZ02]. This means, we use the Fujaba CASE tool to derive automatic JUnit tests from the scenario models. Then, we provide class diagram declarations for all kinds of objects, attributes, links and methods that are used in the scenarios. Next we follow the practical guidance of the Fujaba process and derive (manually) method behavior specifications. Using the automatic tests derived from the scenarios, the methods employed in these scenarios are validated. At the coarse grained level this results in a simulation platform for top-level processes.

Refining the simulation of employed components: Provided with a simulation platform for higher level processes, now certain components may be refined, e.g. the lifter component. We again employ the usual FUP process to derive tests and implementation of the fine grained simulation. Once the fine grained model is unit tested, it may be plugged into the

higher level simulation in order to simulate it in the context of the overall process. Note, different fine grained components may be developed by different teams, concurrently. The overall simulation always serves as a functional reference architecture allowing to test the fine grained component in the overall process and interplay of multiple fine grained component simulations.

Distribution of components: The final system will employ multiple embedded controllers responsible for different components of the overall system. Thus we have to decide which components will be deployed on which devices. We split this into two steps. First we distribute the overall process on multiple independent processes. Then we deploy these process on their target devices. Distributing the overall simulation on multiple process already involves the problem of inter process communication. Thus we have to decide on process communication mechanisms like CORBA or RMI, etc. In addition, the data has to be distributed on the multiple process such that each process has all its information at hand. Maybe parts of the data may have to be replicated and (mutual) update mechanisms have to be installed. Again we use the simulation of the overall system in order to validate process communication and distribution.

Deployment on embedded controllers: Until now, the whole simulation may be validated on a single computer. Once the overall system has been split into multiple components we may deploy the components on different computers. This allows to validate the bus and communication infrastructure. The different components may be deployed on their actual target devices. However, if we still run it in simulation mode, sometimes a more powerful variant of a target embedded controller may be needed to cover the simulation overhead.

Migrating to physical components: Once a component has reached its target embedded controller, we may replace simulation components with their physical counterparts. Our simulation architecture allows to do this in small groups, e.g. one motor and the corresponding sensors. Still, after each reconfiguration we may validate the component using the existing unit tests or employ the component in the context of the overall system. This allows to deal with sensor and actor and IO problems, step by step.

Using the simulation for monitoring purposes: Once the system or a component is successfully deployed in its target configuration, we may still employ the simulation components in order to monitor the behavior of the physical components. We just continue to deploy the simulation and add a monitor component that compares the sensor signals delivered by the simulation with the signals delivered by the physical sensors. A difference beyond a certain threshold may be used as an indicator for malfunctions (either of the simulation or of the physical devices).

4 Summary

In this paper, we introduced a process for developing embedded systems. We think, that in complex embedded systems consisting of several distributed components, an overall system model is needed for testing and simulation purposes. The Fujaba CASE tool already offers tool support for modeling, simulating and testing of such system models. However,

this simulation is still done on the development machine. In section 3, we discuss how the different components may then be deployed to their target platforms. Tool support for this step is future work in the Fujaba CASE tool project. One could e.g. use UML deployment diagrams to specify the distribution of the different components. Some kind of automated CORBA or RMI stub generation for inter-process communication would be desirable, too.

Our process does not yet take hard-realtime requirements into account. Such realtime aspects require additional means for simulation as offered e.g. by MatLab Simulink. To meat the realtime requirements even in the simulation, special hardware may be required. The Fujaba Tool Suite RT developed at the University of Paderborn already enables the developer to model realtime systems using so-called realtime statecharts [FRT04]. From these statecharts, Fujaba then generates JavaRT code and executes it in a simulation VM. Adding this to our process is future work. However, we believe that there are many cases that do not require hard realtime. For such systems, μFUP provides practical guidelines to deal with the distribution and process communication aspects of complex embedded systems deploying multiple communicating micro controllers.

References

[DGMZ02] I. Diethelm, L. Geiger, T. Maier, A. Zündorf: Turning Collaboration Diagram Strips into Storycharts; Workshop on Scenarios and state machines: models, algorithms, and tools; ICSE 2002, Orlando, Florida, USA, 2002.

[DGZ02] I. Diethelm, L. Geiger, A. Zündorf: UML im Unterricht: Systematische objektorientierte Problemlösung mit Hilfe von Szenarien am Beispiel der Türme von Hanoi; in Forschungsbeiträge zur ”Didaktik der Informatik“ - Theorie, Praxis und Evaluation; GI-Lecture Notes, pp. 33-42 (2002)

[Fu02] Fujaba Homepage, Universität Paderborn, http://www.fujaba.de/.

[FRT04] Fujaba Tool Suite RT Homepage, Universität Paderborn, http://wwwcs.upb.de/cs/fujaba/projects/realtime/index.html.

[GSZ03b] L. Geiger, C. Schneider, A. Zündorf: Integrated, Document Centered Modelling in Fujaba; 1st International Fujaba Days, Kassel, Germany (2003).

[KNNZ00] H. Köhler, U. Nickel, J. Niere, A. Zündorf: Integrating UML Diagrams for Production Control Systems; in Proc. of ICSE 2000 - The 22nd International Conference on Software Engineering, June 4-11th, Limerick, Ireland, acm press, pp. 241-251 (2000).

[SGW94] B. Selic, G. Gullekson, P. Ward: Real-time Object Oriented Modeling; ISBN 0471599174, J. Wiley & Sons (1994).

[Zü01] A. Zündorf: Rigorous Object Oriented Software Development, Habilitation Thesis, University of Paderborn, 2001.

A Model-Based Development Process for Embedded Systems

Maritta Heisel[‡] and Denis Hatebur[§]

Abstract: We present a development process for embedded systems which emerged from industrial practice. This process covers hardware and software components for systems engineering, but the main focus is on embedded software components and the modeling of problems, specifications, tests and architectures. Each step of the process has validation conditions associated with it that help to detect errors as early as possible.

1 Introduction

According to Broy and Pree [BP03], about 98% of the CPUs produced worldwide are used in embedded systems. Embedded systems can be found in almost every area of daily life. Moreover, they are often safety- or security-critical. Because embedded systems are usually produced in large numbers, incorrectly functioning systems might cause large damages. Hence, it is crucial to develop embedded systems in such a way that the probability of errors is minimized.

In this paper, we present a development process for embedded systems. That process was developed over time and gradually improved in an industrial context. It is based on development processes used for developing security-critical systems according to the Common Criteria [CC99] and the procedure required for developing safety-critical systems according IEC 61508 [Int98]. The process emerged from projects dealing for example with smartcard operating systems and applets for smartcards in the area of security-critical systems and motor control and automatic doors in the area of safety-critical systems.

The process consists of a sequence of steps to be performed. In each step, a natural-language description or a model (mostly expressed using UML2.0, [OMG03]) is developed. In addition, each step has some *validation conditions* associated with it that help to detect errors as early as possible in the process.

The development process as it is presented in this paper was successfully applied in the development of a protocol converter that connects a proprietary RS-485-based bus system with a CAN-bus system. For this system, there are hard real-time requirements, and the controller has limited memory and performance.

In Section 2, we explain the development process in some detail. An example is given in Section 3. Then, we discuss possibilities for tool support in Section 4. The paper closes with a discussion of the development process (Section 5).

2 Agenda for model-based development

We now present our model-based development process for embedded systems. As a means of presentation, we use the *agenda* concept [Hei98]. An agenda is a list of steps or phases to be performed when carrying out some task in the context of systems and software engineering. The result of the task will be a document expressed in some language. Agendas contain informal descriptions of the steps, which may depend on each other. Agendas are not only a means to guide systems and software development activities. They also

[‡]Universität Duisburg-Essen, Fachbereich Ingenieurwissenschaften, Institut für Medientechnik und Software-Engineering, Germany, email: maritta.heisel@uni-duisburg-essen.de

[§]Universität Duisburg-Essen, Fachbereich Ingenieurwissenschaften, Institut für Medientechnik und Software-Engineering, Germany, email: denis.hatebur@uni-duisburg-essen.de and Institut für technische Systeme GmbH, email: d.hatebur@itesys.de

support quality assurance, because the steps may have validation conditions associated with them. These validation conditions state necessary semantic conditions that the developed artifact must fulfill in order to serve its purpose properly.

Table 1 shows an agenda that precisely describes how to carry out and validate the all the steps of the development process. In the following, each step is motivated and explained in more detail.

Table 1: Agenda for model-based development

No.	Description	Result	Validation
1.	Describe problem	system mission statement (SM), glossary with definitions and designations, requirements (R), domain knowledge (D), assumptions (A) in natural language and a context diagram (see [Jac01])	in Step 2
2.	Consolidate requirements	set of consolidated requirements (R), distinguished between "need to have" and "nice to have"	$D \wedge A \wedge R$ are consistent; $D \wedge A \wedge R' \Longrightarrow SM$; determine set R' ($R' \subseteq R$) of mission-critical requirements
3.	Decompose problem using D, A and R	set of problem diagrams with associated sets of requirements (R)	consistent with SM and context diagrams of Step 1; all requirements have to be captured
4.	**For all subproblems**: derive specification S using R, D and A	specification S of machine to construct (in natural language)	$D \wedge A \wedge S$ are consistent; $D \wedge A \wedge S \Longrightarrow R$
5.	**For all subproblems**: express system behavior, using specifications from Step 4	sequences of interactions between machine and environment (UML 2.0 sequence diagrams)	- all requirements must be captured - in the charts exactly the phenomena of the problem diagram are used - the direction of signals must be consistent with control of shared phenomena as specified in problem diagram - signals must connect domains as connected in problem diagram
6.	Design system architecture using results of Step 5	- system architecture (UML 2.0 composite structure diagram) - perhaps subcomponents (recursively) - all interfaces between the components (UML interface classes) - technical description of hardware interfaces	- all interfaces must be captured - all subproblems must be captured by at least one component

7.	**For all components**: derive interface behavior using results from Steps 5 and 6	interface behavior of all complex components, expressed as UML 2.0 sequence diagrams (test specification)	consistent with input
8.	**For all software components**: design software architecture using results from Step 6, phenomena of problem diagrams from Step 5 and reusable components from other projects	layered software architecture (UML 2.0 composite structure diagram), interfaces between software components (UML interface classes)	phenomena of problem diagrams are interfaces of the application layer; there must be one hardware abstraction layer for each external interface
9.	**For all software components**: develop specification, using results from Steps 7 and 8	component description consisting of: - component overview description (UML 2.0 class diagram with ports and lollipops) - data types (UML-Class-diagrams) - for all operations: pre- and postconditions (OCL or formulas) - invariants (OCL or formulas) - state machine (UML 2.0 state machine diagram)	consistent with interface behavior, completeness of state machines (implies error-cases for user interaction)
10.	**For all software components**: implement software components and test environment for software, using results from Step 7 for tests and Steps 8 and 9 for machine	test environment and software	run tests
11.	Integrate hardware and software using results from Step 10	system and test environment, including test interfaces	run test with hardware and software

Step 1 of the agenda is a creative process. In contrast to other work, we distinguish between requirements and a mission statement. This helps us to classify the requirements in ”need to have” and ”nice to have”. The system mission statement describes the purpose of the system in general terms. The requirements, in contrast, describe in more detail how the environment will behave after the developed system is integrated in it. The requirements are supposed to be a refinement of the system mission. Domain knowledge consists of facts that are true no matter how the embedded system is built. Assumptions are usually rules how users should behave, but which cannot be enforced[1]. The informal way of description used here is helpful to communicate with customers.

In **Step 2**, the consistency between the system mission and the requirements is checked. In particular, the domain knowledge, the assumptions, and the requirements should not be contradictory, and they should

[1] For more details, see [ZJ97, HS99].

suffice to accomplish the system mission. In most cases, domain knowledge, assumptions and further requirements have to be added to successfully perform the check. If there are requirements that are not needed to show that the system mission is accomplished, then either these requirements are not mission-critical, or the system mission is incomplete. Requirements not being mission-critical can be analyzed to decide if the added value for the customer is higher than the estimated cost to develop feature in question.

In **Step 3**, the problem is divided into subproblems, as described by Jackson [Jac01]. Each requirement must belong to the requirements of some subproblem. The subproblems are represented as *problem diagrams* (see [Jac01]).

In **Step 4**, specifications of all the subsystems to be developed (called *machines* by Jackson) are derived. Specifications are implementable requirements. Requirements that are not implementable are transformed into specifications using domain knowledge and assumptions. For an example, see [JZ95]. The specification is a description of the machine that contains all necessary information for its construction. It must be shown that, when the machine fulfills S, then the requirements are satisfied. For that proof, domain knowledge and assumptions can be used.

Step 5 uses the problem diagrams from Step 3 and the specifications from Step 4. For each subproblem, the desired behavior of the corresponding machine is specified using sequence diagrams. For the machine and for each domain in the problem diagram, one lifeline is included in the sequence diagram. The asynchronous signals between the lifelines are annotated with elements of the specified phenomena. This step is equipped with various validation rules that can be used to check the consistency between the problem diagrams an the sequence diagrams.
Experience from many projects has shown that sequence diagrams can easily be discussed with managers and customers that do not have technical knowledge. Loops, states, references and coregions do not cause any problems, while the other new constructs of UML 2.0 such as parallelism, continuation and considered signals should be used with care. The specifications developed in this step can be used as a basis for manual tests.

In **Step 6**, the system architecture is designed. The architecture of the embedded system is expressed as a composite structure diagram. This diagram uses objects for the components, whose ports are connected as described in [OMG03]. The connections are used to transmit the signals of the annotated interfaces between the components. The interfaces with their signals are specified using interface classes. The architecture can be specified recursively, i.e., components can have their own architecture, consisting of sub-components. The external interfaces of the components have to cover the interfaces of all problem diagrams. The architecture must cover all specifications developed in Step 5. This architecture is the starting point for the further development (hardware- as well as software development).

Step 7 refines the sequence diagrams from Step 5 for all complex components of the system architecture. Here, the signals specified in the interfaces of the architecture are used to annotate the sequence diagrams. These sequence diagrams are a concrete basis for the test implementation for all software components.

In **Step 8**, the software architecture for all components containing software is designed. The architecture of embedded software should be a layered architecture. The lowest layer is the *hardware abstraction layer*. This layer covers all interfaces to the external components in the system architecture and provides access to these components independently of the used controller or processor. For porting the software to another hardware platform, only this part of the software needs to be replaced.
The hardware abstraction layer is used by the *interface abstraction layer*. This layer provides an interface that includes the monitored and controlled variables (see [DLP95]) of the system. These variables can be derived from the context diagram or the problem diagrams. It is possible that these variables have to be calculated from the values of several hardware interfaces. For safety-critical software components, the interface abstraction layer will usually make use of redundant arrangements of sensors and actuators.
The highest layer of the architecture is the *application layer*. This layer only has to deal with variables from the problem diagram. Therefore, the system requirements can be directly mapped to the software requirements of the application layer, as described by Bharadwaj and Heitmeyer [BH99].
The software architecture is expressed as a composite structure diagram. To perform this step, the components specified in Step 9 of other projects can be reused.

In **Step 9**, the software components are specified as classes, taking a white-box view. These specifications have to be consistent with Step 7 with respect to the behavior of data types and state machines. The state machines must be complete, i.e., there must be a specified reaction to each possible input signal. The specifications must have the same interfaces as in the component diagram designed in Step 8. In this step, we also have to decide if the component is an active (e.g., behaves like hardware) or passive (e.g., calculation-routine) component. The result of this step forms the basis for the implementation phase.

In **Step 10**, the test environment for all software components is implemented, using the test specification from Step 7. In addition, time frames must be be added, specifying when an event is expected to occur. The system components are implemented using the results of Step 9, using some simple heuristics. The components have to be connected as specified in Step 8. For embedded systems, usually a static connection between components is established. This agenda allows to develop statically linked software components with the capability of reuse. To validate the results of this step, tests may be run in an emulation environment.

In **Step 11**, hardware and software components are integrated. The test of the whole embedded system, consisting of hardware as well as software, is performed.

3 Example: Traffic Light Control

We illustrate our process with the example of a traffic light control system. The system controls the traffic lights for a crossing with a main and a secondary road. In the waiting area of the secondary road, a sensor (more concretely: an induction loop) is integrated that send that sends a signal to the control system to notify it when a car is waiting. The mission statements for the system are as follows:

SM1: The traffic lights should prevent accidents on the crossing.

SM2: The traffic lights should arrange for a fair and adapted flow of traffic between the main and the secondary road.

Step 1: Describe Problem Figure 1 shows the context diagram for the traffic lights control problem. Ordinary boxes are *given domains*, whereas the domain *traffic lights control* is the machine domain, i.e., the domain to be constructed. The annotations at the links connecting the domains are phenomena that are shared by the respective domains. This means that one of the domains controls the phenomenon, and the other domain observes the phenomenon.

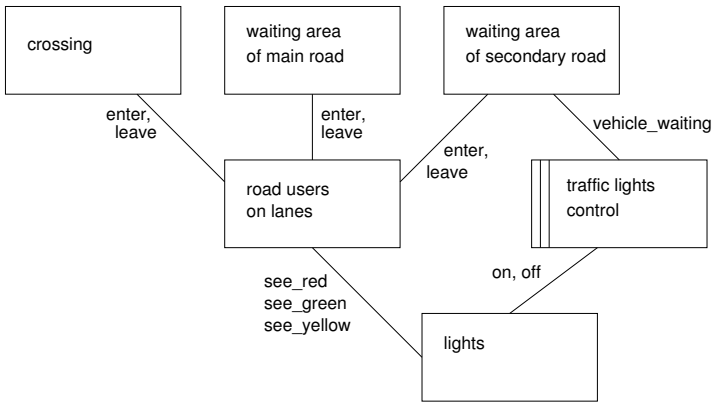

Figure 1: Context diagram for the traffic lights control problem

We have the following requirements:

R1: When there is a car waiting on the secondary road, the traffic lights should stop the flow of traffic on

the main road for a period of time and allow the traffic flow on secondary road.

R2: Vehicles on the main road should be allowed to pass the crossing for at least twice as long as vehicles on the secondary road.

R3: While vehicles on one road are allowed to pass the others should be stopped.

R4: The lights should switch in the following order: red - red/yellow - green - yellow - red. Other combinations are not allowed.

R5: After switching to red, the traffic flow of both roads should be stopped for a period of time.

Step 2: Consolidate Requirements An analysis of the requirements reveals that requirements R3, R4, and R5 are necessary and sufficient to achieve the mission statement SM1, i.e., to avoid accidents. For that demonstration, it is necessary to consider domain knowledge and assumptions that we cannot present here for reasons of space.

Requirements R1 and R2 (together with domain knowledge) are necessary and sufficient to achieve mission statement SM2, i.e., to guarantee a fair and adapted flow of traffic. Hence, all requirements are necessary and must be implemented.

Step 3: Decompose Problem Problem decomposition yields two subproblems, corresponding to the two mission statements. The corresponding problem diagrams are shown in Figures 2 and 3.

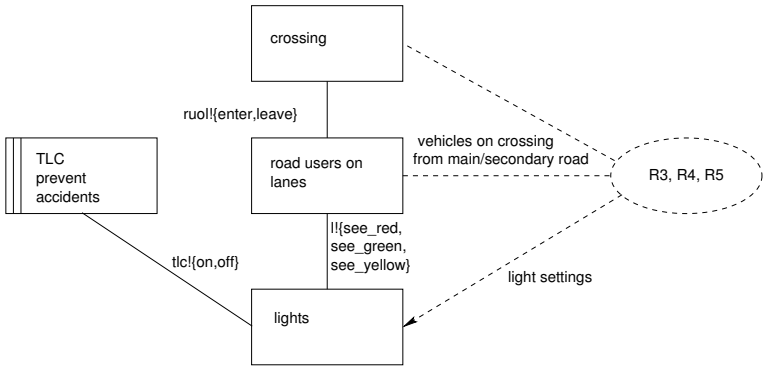

Figure 2: Subproblem of traffic lights control problem: prevent accidents

A problem diagram is distinguished from a context diagrams as follows: first, it states what domain is in control of shared phenomena. For example, the traffic light control domain can switch on and off the lights, which is indicated by "tlc!{on, off}". Second, the requirements to be achieved by the problem are shown in dashed oval. A dashed line from the requirements to a domain indicates that the requirements refer to the domain. A dashed arrow indicates that the requirements constrain the domain in question.

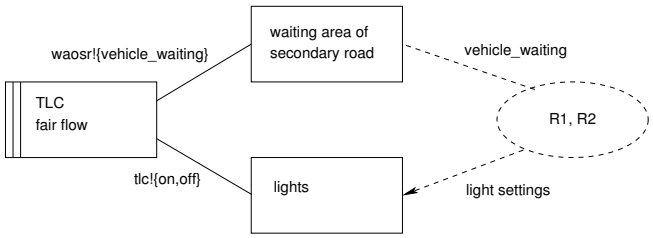

Figure 3: Subproblem of traffic lights control problem: fair flow of traffic

In the traffic lights example, the requirements control the *lights* domain, i.e., the signaling shown by the traffic light system.

Step 4: Derive Specifications According to Jackson and Zave, [JZ95], specifications are *implementable* requirements. A requirement is *not* implementable if it refers to phenomena not observable by the machine, if it constrains phenomena not controlled by the machine, or if it constrains the future. To transform non-implementable requirements into specifications, domain knowledge and assumptions are used. In our example, all requirements are implementable, and thus specifications, because the traffic lights are under control of the machine, and the phenomenon *vehicle_waiting* is observable by the machine.

Step 5: Express System Behavior We now have to specify when and in which order the phenomena shared by the machine with its surrounding domains occur. We give two examples of the corresponding sequence diagrams. The first diagram in Figure 4 states how the system achieves the transition from the state *MAIN_PASSING* to the state *SEC_PASSING*. Another sequence diagram specifies the transition from the state *SEC_PASSING* to the state *MAIN_PASSING*.

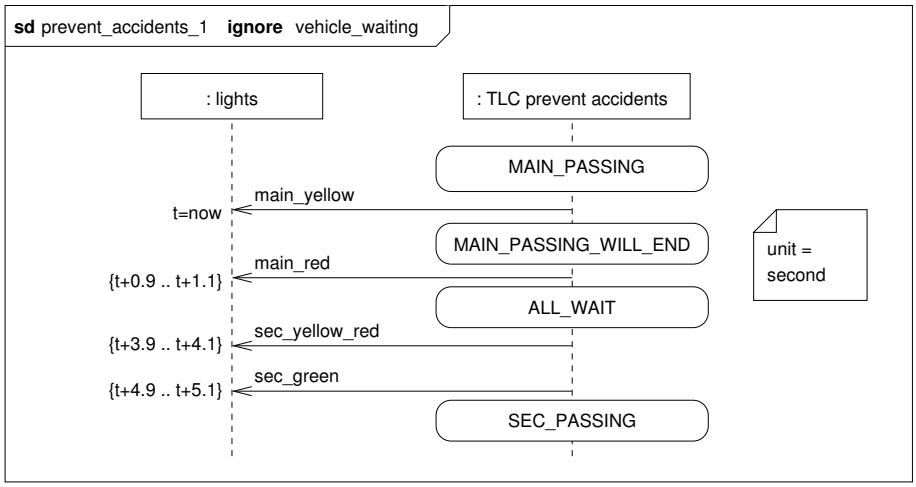

Figure 4: Sequence diagram for subproblem “prevent accidents”

A sequence diagram for the subproblem “fair and adapted flow of traffic” is given in Figure 5. It treats the case where a vehicle on the secondary road is waiting, but the request cannot be taken into account immediately, because the main road must be allowed to pass for at least 20 seconds.

Step 6: Design System Architecture The system architecture shown in Figure 6 consists of a software component *TrafficLightsControl*, which decides on the signaling shown by the physical traffic lights, and two hardware components *LightsControl* (which connects the software to the physical lights) and *InductionLoopControl* (which connects the software to the induction loop).

The interfaces between the components are described by interface classes that contain the signals that can be exchanged via the interfaces. In our example, we have to refine the abstract signal *main_yellow*, *main_red* (see Figure 5) etc. used in Step 5 to concrete signals needed to control the physical traffic light elements. For example, the abstract signal *main_red* is refined to the sequence of signals *main_red(24)*, *main_yellow(0)*, *main_green(0)*. This means that each light bulb is controlled separately, and switching a light bulb on means a volt value of 24V, whereas switching a light bulb off corresponds to a volt value of 0V.

The signals used by the software component *traffic lights control* are more abstract, they have a boolean parameter for each light that indicates if it must be switched on or off. Figure 7 shows the interface classes *lights_on_off* and *lights_on_off_if* that contain the signals described above.

Step 7: Derive Interface Behavior In Step 7, internal behavior of the system is expressed by sequence diagrams describing the order and timing of signals that are sent over the internal interfaces. That behavior

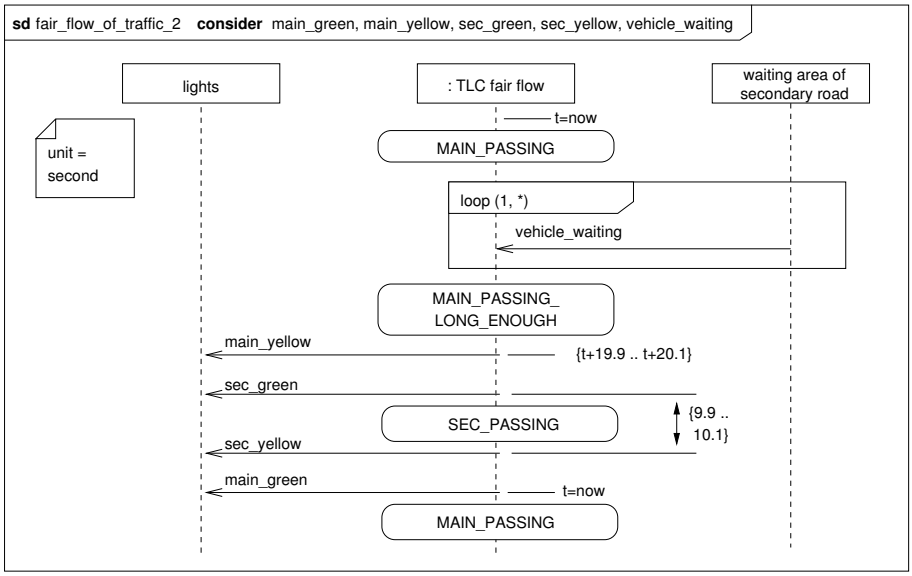

Figure 5: Sequence diagram form subproblem "fair and adapted flow of traffic"

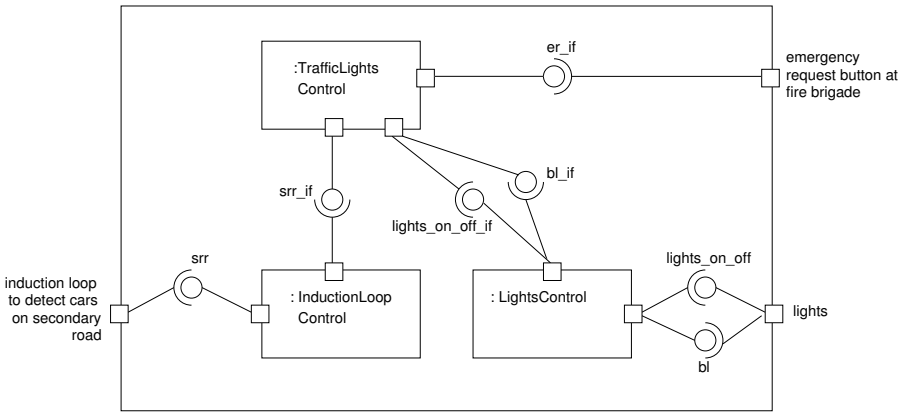

Figure 6: System Architecture for Traffic Lights System

is a refinement of the global system behavior that was specified in Step 5. The signals used are the signals of the interface classes specified in Step 6. Figure 8 shows the sequence diagram that refines the diagram of Figure 4.

Step 8: Design Software Architectures We now have to design the architecture of the software component *traffic lights control* as a layered architecture as described in Section 2. The result is shown in Figure 9. An interface abstraction layer is only needed for the interface of the software component with the hardware component *lights control* (see Figure 6). However, driver components making up the hardware abstraction layer are needed for all software/hardware interfaces.

The component *TrafficLightApplication* has to be refined into a *timer* component that generates timeouts and a component *TrafficLightBehavior*.

Step 9: Develop Software Component Specifications In Step 9, we specify each software component in detail. In order to support reuse, we define a class for each software component, showing its interfaces using the socket/lollipop notation. If the class uses a timer or works in parallel with other components, we

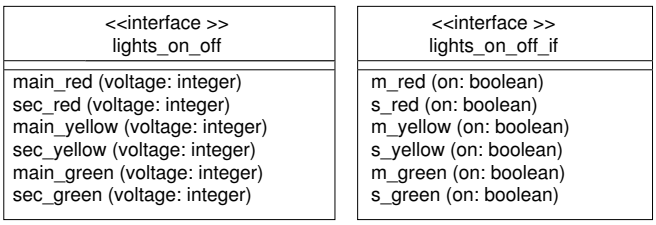

Figure 7: Interface classes for the traffic light system

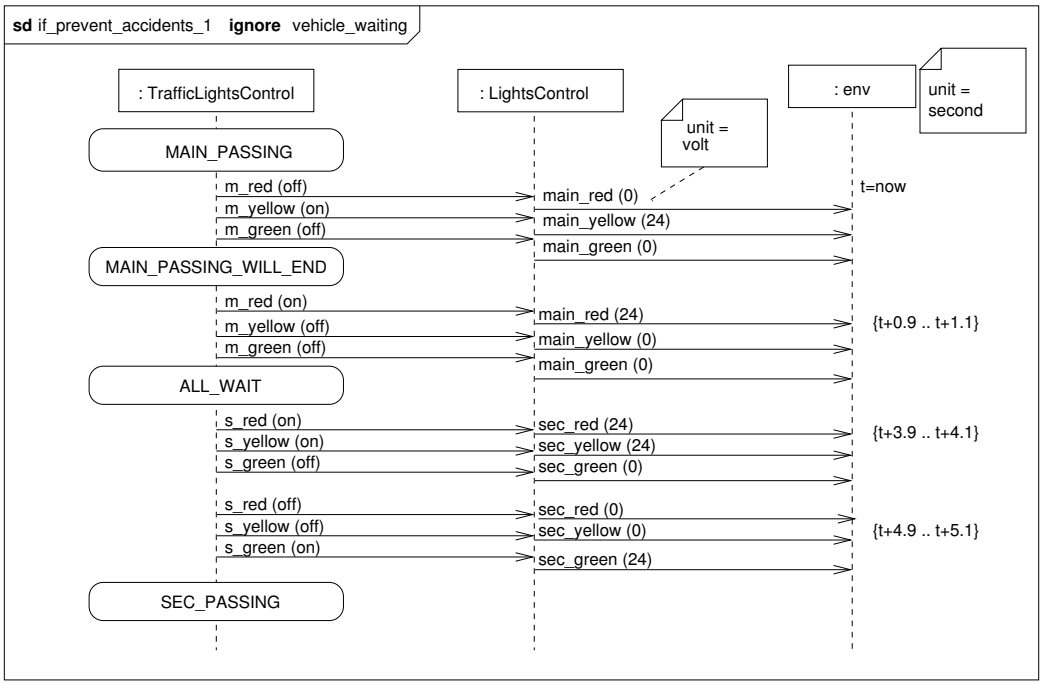

Figure 8: Interface behavior for subproblem “prevent accidents”

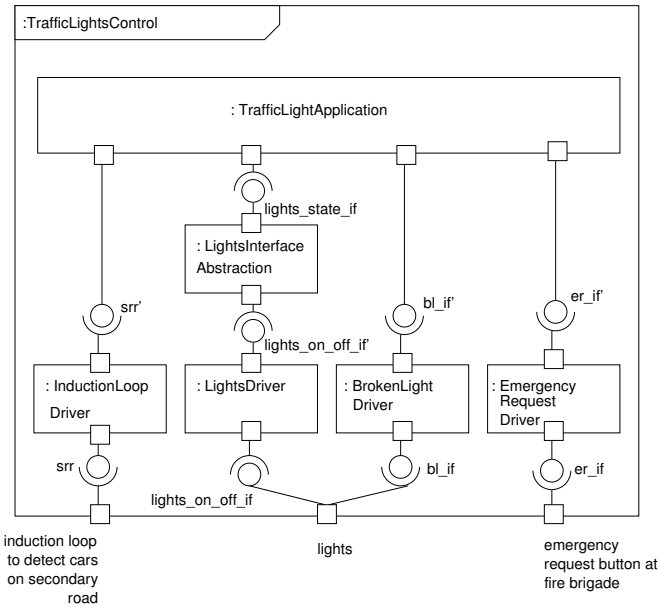

Figure 9: Software architecture for traffic lights control component

use an active class, as shown in Figure 10.

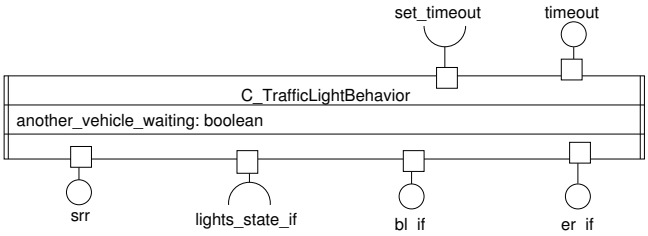

Figure 10: Traffic lights control component overview specification

This class contains an attribute to store if there are are unhandled requests from vehicles on the secondary road. There are no operations in addition to those in the interface classes.

The behavior of the class is described with state machines. Figure 11 shows the composite states *main_phase* and *sec_phase* that are refined in additional diagrams. The system is in the state *main_phase* when the main road may pass and in state *sec_phase* when the secondary road may pass.

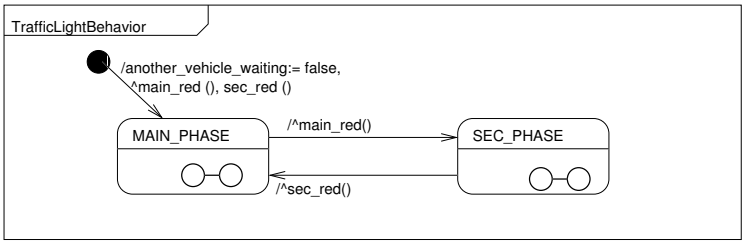

Figure 11: Top-level state machine for traffic lights control component

The state *main_phase* is refined in Figure 12. In this refined state machine, the signals *Timeout* and *srr* are handled and lead to state transitions. The attribute *another_vehicle_waiting* of the class is set when the signal *srr* arrives to store that a vehicle is waiting on the secondary road. That information is then used to decide which of two alternative transitions to take.

This concludes the presentation of the example, as the two remaining phases are concerned with implementation and testing.

4 Tool Support

We plan do equip the process described in Section 2 with tool support. To this end, models developed with specification tools must be exported to be used by validation tools. In particular, we started to extend the free specification tool ArgoUML with the new UML 2.0 composite structure diagram. The standardized XMI file format will then be used to check the consistency between several models created during the development process:

- Steps 6 and 7: It will be checked if the events in the sequence diagram are exactly those specified in the interfaces of the architecture.
- Steps 8 and 9: The interfaces of the architecture and those in the overview specification of each component must be the same.
- Step 9: Only those events specified in interfaces and the operations of the data types are allowed to be used in the state machine diagram.

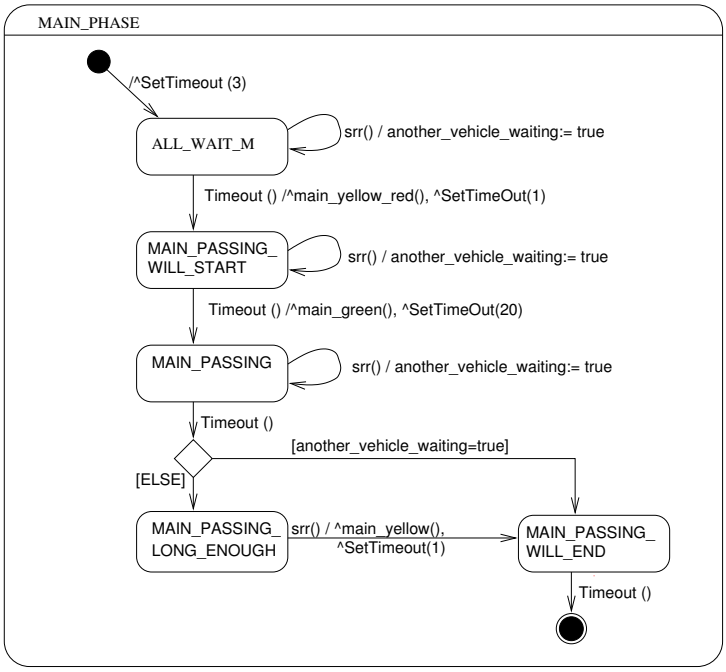

Figure 12: Lower-level state machine for traffic lights control component

We also intend to further enhance our process by using formal methods. Then, it should be possible to export the models to formal verification tools such as Atelier B, FDR, SPIN or SVM. For hardware-software-codesign, export from and to VHDL is planned.

5 Discussion

We now recall the most important characteristics of our development process for embedded systems.

The process proposed here is **model-based**. Modeling is used for problems, specifications, architecture and component behavior. Consistency checks between the several views of the machine are possible (independently from the used tool), because UML provides a standardized XML-based file format that can be parsed easily.

The process covers not only software but the whole system, consisting of **software and hardware**. Within the process, the hardware-software-partitioning problem is addressed. System and software are specified using the same notation. Therefore, the specification can be refined on the system level (Step 6) if more behavioral information is required before the hardware-software-partitioning is possible.

The process is **tailored to embedded systems**. The application domains of many embedded systems can be covered by the four-variable-model proposed by Parnas [DLP95]. Apart from the hardware abstraction layer, the four-variable-model is the most important design criterion for the layered architecture proposed in our development process.

The proposed process supports the **reuse of components** already in the specification phase (see Step 8). Reuse can further be supported by using design patterns.

In large parts, the process makes use of **UML 2.0**. UML 2.0 combines the advantages of the widely known UML and the Specification and Definition Language (SDL) that is used for telecommunication protocols. In contrast to UML 1.4, our layered architecture can be expressed adequately with UML 2.0. In contrast to SDL, UML 2.0 allows a much more flexible structure of components that allows better reuse of

components.

The development of **test cases** is an elementary part in our process. The development of test cases is structured, problem-based and requirement-based. The test specifications are expressed as sequence diagrams, and test cases can be derived (or generated) from these diagrams just by replacing points of time with time frames expressing when desired events are expected.

For each step of the development process, we have defined **validation conditions**. These conditions can be checked using reviews and inspections. However, for many of the validation conditions, formal proof or demonstration is also possible.

The process is defined in such a way that **tool support** can be added in a modular way, based on existing tools.

Finally, our process has been developed in an industrial context, and it was **successfully applied in practice** in several projects for developing security- and safety-critical systems.

References

[BH99] Ramesh Bharadwaj and Constance Heitmeyer. Hardware/Software Co-Design and Co-Validation using the SCR Method. In *Proceedings IEEE International High-Level Design Validation and Test Workshop (HLDV 99)*, 1999.

[BP03] Manfred Broy and Wolfgang Pree. Ein Wegweiser für Forschung und Lehre im Software-Engineering eingebetteter Systeme. *Informatik Spektrum*, 18:3–7, Februar 2003.

[CC99] Common Criteria for Information Technology Security Evaluation, 1999. aligns to ISO/IEC 14508:1999, see http://www.commoncriteria.org.

[DLP95] J. Madey D. L. Parnas. Functional documents for computer systems. In *Science of Computer programming*, volume 25, pages 41–61, 1995.

[Hei98] M. Heisel. Agendas – A Concept to Guide Software Development Activites. In R. N. Horspool, editor, *Proc. Systems Implementation 2000*, pages 19–32. Chapman & Hall London, 1998.

[HS99] M. Heisel and J. Souquières. A Method for Requirements Elicitation and Formal Specification. In J. Akoka, M. Bouzeghoub, I. Comyn-Wattiau, and E. Métais, editors, *Proceedings 18th International Conference on Conceptual Modeling, ER'99*, LNCS 1728, pages 309–324. Springer-Verlag, 1999.

[Int98] International Electrotechnical Commission. Functional safety of electrical/electronic/programmable electronic safty-relevan systems - Part 1: General Requrements, 1998.

[Jac01] M. Jackson. *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley, 2001.

[JZ95] M. Jackson and P. Zave. Deriving Specifications from Requirements: an Example. In *Proceedings 17th Int. Conf. on Software Engineering, Seattle, USA*, pages 15–24. ACM Press, 1995.

[OMG03] Object Management Group OMG. UML 2.0 Infrastructure Specification, 2003.

[ZJ97] P. Zave and M. Jackson. Four dark corners for requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, January 1997. Also availble under http://www.research.att.com/~pamela/ori.html#fre.

Integration of auto-coded Field Loadable Software for the Airbus A380 Integrated Modular Avionics (IMA)

Dipl.-Ing. Andreas Trautmann

Airbus Deutschland GmbH, Avionic Systems Department

Kreetslag 10, 21129 Hamburg, Germany

Integrated Modular Avionics (IMA) today stands as a synonym for a modern, highly standardised and flexible avionics technology, especially considering aircraft customisation needs and potential future changes of on-board functionality during the life cycle of an aircraft and its avionics in general. Due to the architectural concept, which provides independent avionics components it is possible to realise a relatively high reactivity to customisation needs and the provision of resource spares allows the quick extension of the system within a pre-defined range. This allows using this technology in a wide range of applications without the restrictions of conventional avionics systems.

On the other hand the usage of the IMA technology and it's respective architecture has not only an impact on the functional extensibility and flexibility of the avionics system itself but also requires to reconsider the industrial approach for the production, the in-service operation and the maintenance concepts of the avionics equipment during the whole life-cycle of the aircraft.

For example the configuration software that defines the resource allocation of the module at runtime and the respective interfaces is field-loadable and certified as Level A software according to RTCA DO-178B when used in safety-critical applications. In the Airbus A380 program this configuration software is produced by the aircraft manufacturer who, as the integrator, exclusively holds the information of all specific resource needs of all the functions that are integrated in the avionics equipment. Airbus specifies the resource allocation (memory, time and interfaces) by compiling this content to the equipment specific configuration software.

In the presentation the development and the major steps of the integration process of the Field Loadable configuration software for the A380 Integrated Modular Avionics as well as the corresponding industrial considerations are being shown.

Technische Universität Braunschweig
Informatik-Berichte ab Nr. 2000-05

2000-05	S. Eckstein, P. Ahlbrecht, K. Neumann	Von parametrisierten Spezifikationen zu generierten Informationssystemen: ein Anwendungsbeispiel
2000-06	F. Strauß, J. Schönwälder, M. Mertens	JAX - A Java AgentX Sub-Agent Toolkit
2000-07	F. Strauß	Advantages and Disadvantages of the Script MIB Infrastructure
2000-08	T. Gehrke, U. Goltz	High-Level Sequence Charts with Data Manipulation
2000-09	T. Firley	Regular languages as states for an abstract automaton
2001-01	K. Diethers	Tool-Based Analysis of Timed Sequence Diagrams
2002-01	R. van Glabbeek, U. Goltz	Well-behaved Flow Event Structures for Parallel Composition and Action Refinement
2002-02	J. Weimar	Translations of Cellular Automata for Efficient Simulation
2002-03	H. G. Matthies, M. Meyer	Nonlinear Galerkin Methods for the Model Reduction of Nonlinear Dynamical Systems
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: Modellbasierte Entwicklung eingebetteter Systeme