Institut für Rechnergestützte Ingenieursysteme
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3126

Erstellung eines intelligenten Systems zur Generierung von Komponenten für formal beschriebene Fahrzeug-Netzwerke

Philipp M. Frank

Studiengang:	Softwaretechnik
Prüfer:	Univ.-Prof. Hon.-Prof. Dr. Dieter Roller
Betreuer:	Dipl.-Inf. Akram Chamakh Dipl.-Inf. Sascha Opletal
begonnen am:	3. Januar 2011
beendet am:	2. Juni 2011
CR-Klassifikation:	C.3, D.1.2, D.2.6, D.2.11

Versicherung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben. Diese Arbeit wurde in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht.

Philipp M. Frank

Kurzfassung

In dieser Arbeit wird ein erweiterbares Framework zur Generierung von Restbussimulationen für Fahrzeug-Netzwerke vorgestellt. Die Anforderungen an ein derartiges Framework werden erörtert und mit bestehenden Architekturmodellen im Gesamtfahrzeug-Kontext kontrastiert. Die Variabilität der Kommunikationsbeschreibungen für Fahrzeug-Netzwerke wird im Generierungsprozess berücksichtigt, so dass ein kontrolliertes Nachziehen von Änderungen ermöglicht wird. Ein Fokus des entwickelten Frameworks liegt in der Erweiterbarkeit hinsichtlich zusätzlicher Beschreibungsformate für die Buskommunikation sowie hinsichtlich zusätzlicher Zielplattformen.

Abstract

This work introduces an extensible framework for generating residual bus simulations of vehicle networks. The requirements for this framework are discussed and contrasted against existing architectural models for representing the software architecture of a vehicle. The variability of the input specifications for a simulation is taken into account throughout the generation process, leading to a maintainable solution. The framework offers extension points for supporting additional input formats for communication specifications as well as for additional target platforms.

Inhaltsverzeichnis

Kapitel 1

Einführung

Die Softwareentwicklung für den Automobilbau stellt die beteiligten Entwickler auf Seiten des Herstellers wie auf Seiten der Zulieferer vor Integrationsaufgaben, die mit den Prozessen und Vorgehensweisen aus anderen Industriebereichen nicht sinnvoll zu bewältigen sind. Dabei liegen die Gründe nicht in der Komplexität der einzelnen Steuergeräte, sondern im hohen und immer mehr zunehmenden Vernetzungsgrad und in der Variabilität, die im Rahmen des Produktlinienentwicklungs-Prozesses für das übergeordnete Gesamtfahrzeug entsteht, sowie in der sehr speziell strukturierten Zuliefererstruktur. Durch sehr hohe Stückzahlen gibt es für Zulieferer kaum Spielraum in der Auslegung der Hardware; die Software muss entsprechend nahe an der Hardware sein, was dem zugleich verlangten Anspruch der Variabilität entgegenläuft. Die Code-Generierung ist ein Ansatz, diese Aspekte in Einklang zu bringen.

1.1 Aufgabenstellung und Übersicht

In dieser Arbeit wird ein Framework zur Generierung von Restbussimulationen für Fahrzeug-Netzwerke vorgestellt, das die geschilderte Problematik im Kontext der Steuergeräte-Entwicklung und des Steuergeräte-Tests aufgreift. Die Anforderungen an Systeme zur Generierung von Restbussimulationen werden erörtert und mit bestehenden Architekturmodellen im Gesamtfahrzeug-Kontext kontrastiert. Die Variabilität der Kommunikationsbeschreibungen für Fahrzeug-Netzwerke wird im Generierungsprozess berücksichtigt, so dass ein kontrolliertes Nachziehen von Änderungen ermöglicht wird. Ein Fokus des entwickelten Frameworks liegt in der Erweiterbarkeit hinsichtlich zusätzlicher Beschreibungsformate

für die Buskommunikation sowie hinsichtlich zusätzlicher Zielplattformen.

Diese Arbeit gliedert sich wie folgt: das vorliegende Kapitel erläutert den fachlichen und technischen Kontext. Kapitel 2 untersucht die Anforderungen an die Software-Architektur von Restbussimulationen und stellt diese der AUTOSAR-Architektur gegenüber. Kapitel 3 stellt die grundlegenden Konzepte bei der Code-Generierung vor und beschreibt das entwickelte Framework im Kontext dieser Konzepte. Kapitel 4 stellt einige Details der Implementierung vor. Kapitel 5 stellt mögliche Erweiterungen und Verbesserungen vor, die auf den Ergebnissen dieser Arbeit aufbauen können.

1.2 Firmenporträt

Die Firma Berger Elektronik GmbH ist ein in Sindelfingen ansässiges Unternehmen im Automobil-Sektor mit den Schwerpunkten Bordbussysteme und Steuerungselektronik, insbesondere für das Prüfstand-Umfeld. Die Berger Elektronik ist an die in Böblingen beheimatete Star Cooperation Gruppe angeschlossen. Intern gliedert sich der Betrieb in Entwicklungsabteilung, Kfz-Werkstatt und Produktions-Werkstatt.

1.3 Kommunikationsnetzwerke und Bussysteme in Fahrzeugen

Modelle moderner Fahrzeugbaureihen im Personen- wie auch im Nutzfahrzeug-Bereich weisen einen hohen Vernetzungsgrad hinsichtlich der in ihnen verbauten Steuergeräte auf. Waren in der Vergangenheit hauptsächlich Anwendungen in der Motor-Steuerung und im Getriebestrang sowie sicherheitsrelevante Aspekte und gesetzliche Anforderungen an die Umweltverträglichkeit verantwortlich für einen starken Anstieg der notwendigen Konnektivität, so zählen heute auch und vor allem Komfort-Funktionen und Infotainment-Systeme zu den treibenden Kräften für neue Funktionalität und damit verbundene neue Technologien.

Im Folgenden werden die wichtigsten Bussysteme (CAN, LIN und Flexray) kurz beschrieben; für eine detaillierte Beschreibung, insbesondere im Hinblick auf die physikalische Topologie, die in der vorliegenden Arbeit eine sehr untergeordnete Rolle spielt, sei auf die einschlägige Literatur verwiesen (insbesondere [Par07], [ZS10] und [Rei08]).

CAN (*Controller Area Network*) realisiert einen Multi-Master-Bus, d.h. alle teilnehmenden Steuergeräte sind gleichberechtigt und können prinzipiell zu

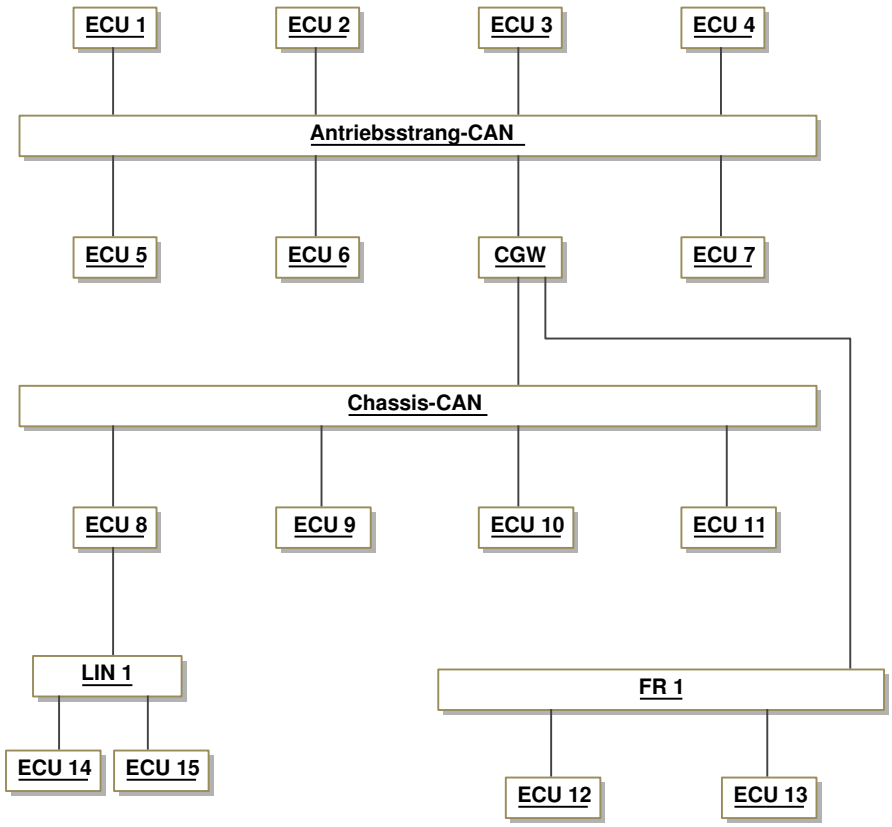

Abbildung 1.1: Typische Topologie eines heterogenen Bordnetzwerkes [Ber]

beliebigen Zeitpunkten Sendeversuche starten, sofern der Bus frei ist. Die Adressierung erfolgt inhaltsbasiert mittels des Identifiers einer versendeten Nachricht. Dieser Identifier dient auch als Basis für die Arbitrierung, wobei CAN eine bitweise Arbitrierung definiert, die bei Konflikten diejenige Nachricht mit dem kleinsten Identifier am höchsten priorisiert, so dass der Sender dieser Nachricht den Zugriff auf den Bus erhält; dies ist im Wesentlichen eine Variante von CSMA/CA. Der korrekte Empfang von Nachrichten wird überprüft, indem jeder Empfänger auf dem Bus eine empfangene Nachricht bestätigt; hierzu setzt jeder Empfänger, der eine Nachricht korrekt erhalten hat, ein Empfangs-Bit unmittelbar beim

Empfang der Nachricht. Dies wird vom Sender erkannt; bleibt eine Empfangsbestätigung aus, d.h. setzt keiner der potentiellen Empfänger das Empfangs-Bit oder ist kein Empfänger vorhanden, so definiert CAN ein bestimmtes Verhalten für Wiederholungs-Versuche und den Übergang in Fehlerzustände [Par07].

LIN (*Local Interconnect Network*) ist als einfaches, kostengünstiges Bus-System für räumlich beschränkte Sensor-Aktor-Netzwerke ohne hohen Bandbreitenbedarf und mit begrenzter Teilnehmerzahl konzipiert worden, mit dem Hintergrund, diesen Anwendungsfall günstiger abdecken zu können als es durch ein vergleichbares CAN-Netzwerk möglich wäre [ZS10]. Ein LIN-Netzwerk besteht aus einem Master-Steuergerät und einem oder mehreren Slave-Steuergeräten, wobei nur der Master die Kommunikation initiieren darf. Der Master steuert den Buszugriff, indem er den Slaves Botschaften sendet, auf die diese reagieren können, wobei in der Schedule definiert ist, welches Slave-Steuergerät auf welche Botschaften antworten darf; das Zeitverhalten, das ebenfalls in dieser Schedule definiert ist, ist nur für den Master relevant. LIN definiert besondere Nachrichten, die für die Initiierung von Sleep- und Wakeup-Verhalten verantwortlich sind und legt zeitliche Reaktionsfenster fest, die von Master und Slaves beim Eintritt in den Sleep-Modus und beim Wiedereintritt in den aktiven Modus eingehalten werden müssen. Erwähnenswert ist, dass die LIN-Spezifikationen [LIN] nicht nur die Übertragungsschicht und das Protokoll beschreiben, sondern auch ein generisches API und ein Format zur formalen Beschreibung von LIN-Netzwerken (siehe Abschnitt 1.4) festlegen.

Die mangelnde Echtzeitfähigkeit von CAN sowie die fehlende Möglichkeit, auf Protokollebene redundante CAN-Netzwerke einzurichten (beides Voraussetzung für so genannte X-By-Wire-Anwendungen[1]) führte zur Entwicklung des Byteflight-Protokolls, das als Flexray-Protokoll in einer modifizierten Ausprägung standardisiert wurde [ZS10]. Echtzeitfähigkeit wird hier durch die Verwendung von TDMA und einer Scheduling-Tabelle sowie Mechanismen zur globalen (d.h. innerhalb eines Netzwerk-Clusters) Synchronisation der Zeitgeber aller Cluster-Teilnehmer ermöglicht. Die Ausfallsicherheit wird (optional) erhöht, indem zwei physikalisch getrennte Kanäle für die redundante Übertragung von Informationen genutzt werden, wobei die Konsistenzprüfung durch die Bus-Controller auf Protokollebene erfolgt und nicht durch ECU-Software auf Anwendungsebene

[1] Ein Überbegriff für Steer-By-Wire, Brake-By-Wire und ähnliche Mechanismen. Im KFZ-Kontext wird damit die Ersetzung von physikalischen Steuerungseinrichtung wie Lenksäule und Bremsleitung durch zuverlässige und ausfallsichere Bussysteme bezeichnet. Zwar stehen heute bereits alle Informationen als Sensorwerte zur Verfügung, um dies zu realisieren (Lenkradwinkel und Bremspedalstellung), jedoch ist es gesetzlich nicht erlaubt, vollständig auf die physikalischen Übertragungswege zu verzichten.

erfolgen muss.

Der Bereich Infotainment, d.h. der Bereich der Integration von Unterhaltungselektronik in das Bordnetzwerk von Personenkraftfahrzeugen der Oberklasse, war darüber hinaus Motivation für die Entwicklung eines komplett neuen Bussystems (MOST, Media Oriented Systems Transport), das dem im Vergleich zu herkömmlichen Anwendungen extrem erhöhten Bandbreiten-Bedarf Rechnung trägt [ZS10].

1.4 Beschreibungsformate für Buskommunikation

Zur sinnvollen Auslegung der Bus-Systeme für das Bordnetzwerk ist es notwendig, die gesamten möglichen Kommunikationspfade und -inhalte auf der Ebene des Gesamt-Systems zu erfassen. Zu diesem Zweck, d.h. zur formalen Beschreibung der Kommunikations-Muster in Bordnetzwerken, existieren verschiedene Standards und Industrie-Standards, die ein Datenmodell und ein Dateiformat (teilweise auch eine Methodologie) definieren, die dies ermöglichen.

Diese Beschreibungsdateien spielen im Systemintegrations-Prozess zwischen Gesamtfahrzeug-Hersteller und Zulieferern eine zentrale Rolle; in Abschnitt 2.6.1 wird dies (im Kontext der AUTOSAR-Methodologie) näher erläutert.

Das Datenmodell, das durch diese Beschreibungsdateien beschrieben wird, enthält als zentrale, in allen Formaten vorhandene Elemente die folgenden Elemente:

Busse und deren Parameter Die physikalisch vorhandenen Busse, deren Typ und deren Parameter müssen definiert werden können. Für den CAN-Bus sind dies die Bitrate und der Typ, für LIN und Flexray sind zusätzliche Parameter für die Einrichtung und Aufteilung der Scheduling-Tabelle notwendig.

ECUs ECUs (*Electronic Control Unit*) entsprechen (physikalisch vorhandenen) Steuergeräten und müssen beschrieben werden können.

Netzwerkknoten Verwendet eine ECU einen bestimmten physikalisch vorhandenen Bus, so ist sie ein Netzwerkknoten auf diesem Bus. Ist eine ECU an mehrere Busse angebunden, so existiert sie als logischer Netzwerkknoten auf allen diesen Bussen. Logisch entspricht dies einer Zuordnung von ECUs zu den physikalisch vorhandenen Bussen.

Nachrichten auf Protokoll-Ebene Die auf jedem Bus ausgetauschten Nachrichten sowie deren busspezifische Parameter müssen definiert werden können.

Für den CAN-Bus ist dies relativ einfach möglich (zu jeder Nachricht muss lediglich der CAN-Identifier und die Länge definiert werden können). Für Flexray ist die Zuordnung einer Nachricht zu einem Slot in der Schedule notwendig. Darüber hinaus ist es in der Regel so, dass für den Nachrichten-Inhalt bestimmte Default-Belegungen angegeben werden können oder Bitmuster für nicht verwendete Teile einer Nachricht definiert werden können.

Signale Signale entsprechen in der Regel bestimmten Werten aus dem physikalischen Fahrzeug-Modell, z.B. Sensorwerten, oder Notifikationen über Ereignisse, die im Fahrzeug auftreten.

Signal-Kodierung Die konkrete Darstellung von Signalen muss in einem bestimmten Format (Wertetyp, Bitlänge usw.) erfolgen; diese Kodierungs-Informationen müssen definiert und den Signalen zugeordnet werden können. Eine Umrechnungs-Vorschrift von dieser Darstellung in konkrete physikalische Werte muss ebenfalls definiert und einem Signal zugeordnet werden können. Näheres zu dieser Thematik ist in Abschnitt 2.1.2 beschrieben.

Zuordnung von Signalen zu Nachrichten Für Signale muss definiert werden können, in welchen Nachrichten sie in welcher Kodierung vorkommen und welche Teile im Byte-Array der Nachricht sie belegen.

Zuordnung von Signalen zu Empfängern Für Signale muss definiert werden können, welche Netzwerkknoten sie empfangen. Hier stellt sich die Frage, warum dies nicht auf Nachrichtenebene erfolgt, da dies dem Verhalten auf Protokollebene entspricht. Die Antwort liegt darin, dass durch die explizite Beschreibung, welche Signale von einem Knoten empfangen werden, eine höhere Flexibilität erreicht wird, da dann ein Wechsel der Nachricht, die das Signal enthält, möglich ist, ohne dass dies notwendige Anpassungen nach sich zieht. Darüber hinaus ermöglicht dies eine Optimierung hinsichtlich des Nachrichteninhaltes, indem statisch ein Filter für den Bus-Controller der empfangenden ECU definiert werden kann, der nur bei Änderungen der tatsächlich für diese ECU relevanten Teile der Nachricht eine Weiterleitung der Nachricht vom Bus-Controller zum eigentlichen Host-Microcontroller veranlasst.

Zuordnung von Nachrichten zu Sendern Zu jeder Nachricht muss definiert werden können, von welchem Netzwerkknoten sie gesendet werden darf.

Bei neueren Beschreibungsformaten kommt eine zusätzliche Abstraktionsebene zwischen Signalen und Botschaften hinzu, die so genannten PDUs (*Protocol Data Unit*) [AUTa]. Signale verweisen dann nicht direkt auf Botschaften, sondern auf PDUs, und diese verweisen auf Botschaften. Hintergrund und Motivation dafür ist, dass unterschiedliche Bussysteme jeweils andere Maximalgrößen für Botschaften definieren. Die PDUs dienen dazu, die Einhaltung dieser Maximalgrößen zu gewährleisten, ohne auf die Verwendung großer Botschaften auf Bussystemen, die diese unterstützen, zu verzichten [AUTb].

Das älteste im größeren Umfang eingesetzte und derzeit noch am weitesten verbreitete Format ist das von der Firma Vector Informatik entwickelte und durch eine umfangreiche Werkzeugkette unterstützte DBC-Format (das Akronym steht für den generischen Begriff *Database Container*), das auf die Beschreibung von CAN-Netzwerken beschränkt ist. Die Serialisierung erfolgt als textbasierte Datei, deren Syntax in keinerlei Weise standardisiert oder vollständig öffentlich definiert ist und die üblicherweise mit dem Werkzeug *CanDB* von Vector Informatik erstellt wird. Das DBC-Format ermöglicht die Definition beliebiger Attribute und die Zuordnung dieser Attribute zu den Modell-Elementen des Bordnetzes, so dass Herstellern die Definition von eigenen, über die Grundfunktionalität hinausgehende semantische Erweiterungen ermöglicht wird, sofern sich diese Erweiterungen durch die Definition von Attributen für Elemente realisieren lassen. Diese Attribute sind typisiert, so dass rudimentäre werkzeuggestützte Konsistenzprüfungen möglich sind. Eine typische Anwendung für dieses Attribut-Metamodell ist die Definition von Attributen, die von nachgelagerten Werkzeugen der Werkzeugkette verwendet werden können, z.B. von Code-Generatoren.

Spezifisch für LIN existiert das LDF-Format (*LIN Description File*), das Teil der LIN-Spezifikationen ist (ergänzt durch so genannte *Node Capability Files* (NCF)), die zusätzliche, ECU-zentrische Sichten realisieren).

Eine neuere, durch eine herstellerübergreifende Organisation gestützte Spezifikation existiert unter dem Namen Fibex (*Fieldbus Exchange Format*), die eine XML-basierte serialisierte Darstellung definiert und generell nicht busspezifisch ist, sondern die Bus-Systeme CAN, LIN, Flexray und MOST unterstützt, wobei kleinere Teil-Spezifikationen für notwendige busspezifische Spezialisierungen als gesonderte XSD-Definitionen existieren [ZS10].

Auch AUTOSAR enthält Mechanismen zur Beschreibung von Bordnetzwerken, wobei sich diese semantisch an dem von Fibex vorgeschriebenen Modell orientieren (in der AUTOSAR-Terminologie wird dies *Fibex-Core* genannt) [AUTd].

1.5 Restbussimulationen

Der hohe Vernetzungsgrad hat Auswirkungen auf die Entwicklung und den Test von Steuergeräten und Fahrzeug-Komponenten, da es dadurch in aller Regel so ist, dass das zu testende Steuergerät nicht in Isolation (d.h. ohne die Präsenz anderer Steuergeräte an einem der Bussysteme, die das zu testende Steuergerät verwendet) getestet werden kann [ZS10].

Die zu testende ECU erwartet bestimmte Nachrichten, die in bestimmten Zeitintervallen auftreten müssen, um nicht in einen Fehlerzustand überzugehen und die Funktion einzustellen. Es kann hierbei unterschieden werden zwischen Nachrichten, die tatsächlich notwendig für die Funktion des zu testenden Steuergerätes sind (etwa weil sie Signale enthalten, die von der ECU weiterverarbeitet oder ausgewertet werden), und Nachrichten, die als Kontroll-Nachrichten dienen, um zu erkennen, ob das Bordnetzwerk intakt ist. Dies kann weiter unterschieden werden in implizite Kontroll-Nachrichten, d.h. zyklische Nachrichten, die eigentlich konkrete Inhalte enthalten, die aber zur Zustands-Ermittlung des Bordnetzwerkes missbraucht werden und explizite Netzwerk-Management-Nachrichten, die spezifisch zu diesem Zweck existieren. Andererseits erwartet die ECU, sofern sie selbst periodische Nachrichten generiert, dass diese von anderen ECUs bestätigt werden (zum Beispiel durch das Setzen des Acknowledgement-Bits, das im Falle der CAN-Kommunikation auf Protokoll-Ebene anzeigt, ob eine Nachricht von mindestens einem weiteren Bus-Teilnehmer empfangen wurde). Fehlen diese Bestätigungen, so geht die zu testende ECU u.U. ebenfalls in einen Fehlerzustand über.

Der Ansatz, diese Erwartungen der ECU in Bezug auf ihre Umwelt (d.h. die vorhandenen Busteilnehmer) während der Test-Läufe auf der ECU selbst zu simulieren, z.B. durch einen Dummy-Bus-Treiber, scheitert aus verschiedenen Gründen. Zum einen sind Serien-ECUs so knapp dimensioniert, dass derartige Zusatzfunktionalität zu Testzwecken nicht mit untergebracht werden kann. Zum anderen unterliegt der Quellcode für Serien-ECUs strengen Audit-Regeln und ist u.U. formal validiert, so dass jegliche Zusatzfunktionen zu Test-Zwecken diese Validierung untergraben würde (oder ebenfalls auditiert und validiert werden müssten). Ein anderer Aspekt ist, dass das Fehlverhalten bei fehlender Bus-Anbindung oder fehlenden Kommunikations-Partnern ein integraler Bestandteil des von der ECU zu realisierenden Verhaltens ist, so dass dieses ohnehin getestet werden muss, selbst wenn es möglich wäre, Teile der Funktionalität ohne echte Bus-Anbindung zu simulieren.

Daher ist es zum Test von Steuergeräten notwendig, die fehlende Kommuni-

kation zu ersetzen, indem auf dem Bus, auf dem ein Steuergerät Netzwerkknoten ist, eine gewisse Grund-Kommunikation erzeugt wird und so die fehlenden anderen ECUs, die eigentlich zur Funktion des zu testenden Steuergerätes notwendig sind, zu simulieren. Dies wird mit dem Begriff *Restbussimulation* bezeichnet. Hierbei wird unterschieden zwischen der statischen Restbussimulation und der dynamischen Restbussimulation. Die statische Restbussimulation erzeugt Bus-Kommunikation, indem die benötigten Nachrichten (deren Aufbau und Standard-Inhalt z.B. aus eine Beschreibungsdatei ermittelt werden kann) gemäß der ihnen zugeordneten Parameter (Zykluszeit) periodisch gesendet werden. Die dynamische Restbussimulation ergänzt dies um beliebiges Verhalten, d.h. die Nachrichteninhalte und Parameter sind nicht a priori festgelegt, sondern werden in Reaktion auf externe Ereignisse verändert und gesendet. Im Kontext dieser Arbeit wird die dynamischen Restbussimulation als Anwendung betrachtet, die auf ein statisches Restbussimulations-Gerüst zugreift und dieses durch beliebigen benutzerdefinierten Code ergänzt.

Dieses prinzipielle Konzept der Restbussimulation ist in Abbildung 1.2 dargestellt.

Restbussimulationen kommen nicht nur beim Test von einzelnen ECUs zum Einsatz, sondern können im Kontext der Serienfertigung an Prüfständen eingesetzt werden, um Fahrzeug-Komponenten zu testen. Häufiger Einsatzfall sind Motoren-Prüfstände, da das Motorsteuergerät in der Regel von vielen anderen ECUs abhängig ist und damit eine umfangreiche Simulation dieser ECUs für den isolierten Test eines Motors notwendig ist. Hierbei kommen noch zusätzliche Aufgaben hinzu, wie die Anbindung an Systeme zur Vorgabe von Parameterwerten und zur Protokollierung der Prüfungen sowie die Anbindung an Systeme, die die Leistungs-Elektronik steuern und im Fall von Motoren-Prüfständen den Zu- und Abluft-Strom regulieren und protokollieren. Die Anbindung dieser Systeme erfolgt in der Regel ebenfalls über die bereits vorhandenen Busse (nicht zuletzt um die Einführung zusätzlicher Hardware und Verkabelung zu vermeiden).

Ein konträrer, komplementärer Einsatzfall für Restbussimulationen ist der Austausch einer ECU durch eine Restbussimulation im realen Fahrzeug. Dies ist z.B. sinnvoll für den Test neuer Algorithmen für Stabilitäts-Programme und ähnliche aufwändige Anwendungsfälle. Die Implementierung neuer Algorithmen direkt auf der betroffenen ECU ist nicht praktikabel, da dies aufwändige Anpassungen an die Limitierungen der vorhandenen ECU erfordern würde und so ein schnelles, kosteneffizientes Prototyping von Algorithmen nicht möglich wäre.[2]

[2]Dies beruht auf der in der Regel zutreffenden Annahme, dass das eingesetzte System zur Restbussimulation wesentlich mächtiger und einfacher erweiterbar ist als ein Seriensteuergerät.

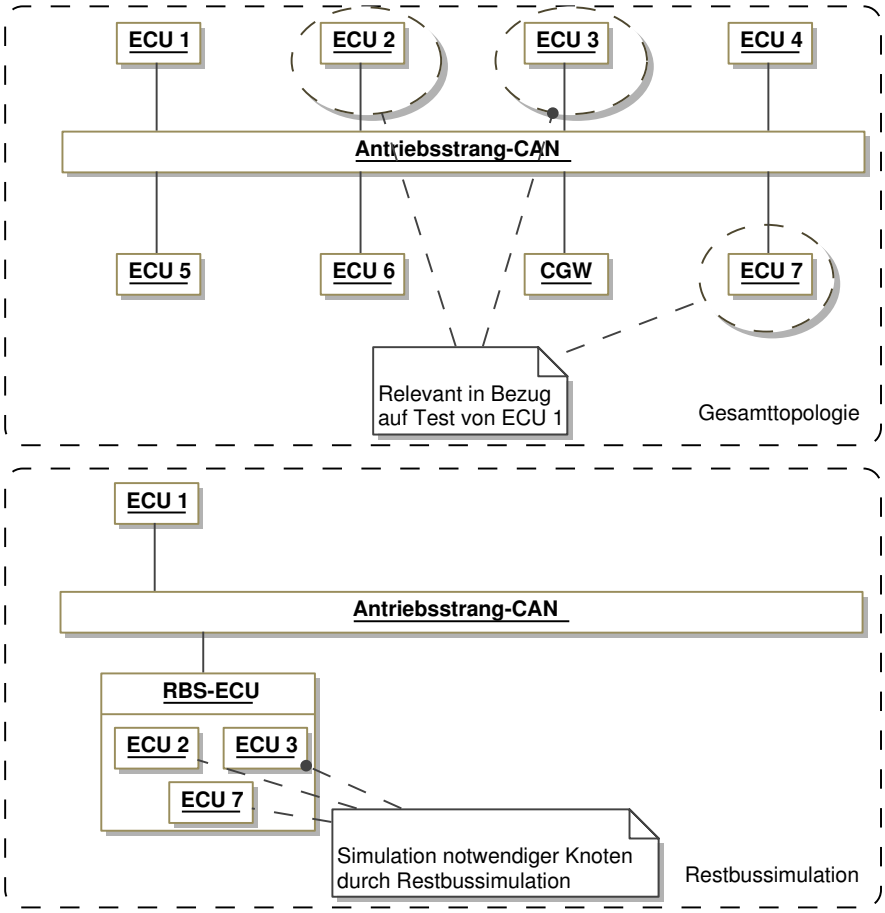

Abbildung 1.2: Klassischer Anwendungsfall der Restbussimulation [Ber]

1.6 Gateways

Mit Gateways werden ECUs bezeichnet, die als wesentliche Aufgabe das Abbilden von Nachrichten oder Signalen von einem Bus auf einen anderen Bus übernehmen, wobei unterschieden wird zwischen Gateways, die zwischen physikalisch unterschiedlichen Bussystemen vermitteln können (so genannte Protokoll-Gateways) und Gateways, die lediglich zur Isolierung unterschiedlicher Busse dienen. Diese Isolierung kann physikalischer Natur sein (z.B. durch Optokoppler) oder logischer Natur, z.B. soll der Diagnose-Bus in der Regel nicht alle Nachrichten mitlesen können und nicht beliebige Nachrichten an die internen Busse

weiterleiten können. Um diese Gateway-Steuergeräte zu simulieren, muss ein Framework für Restbussimulationen Mechanismen zum Routing von Informationen zwischen den Bussen einer ECU bereitstellen.

Auch Steuergeräte, die keine explizite Gateway-Funktion haben, können an mehreren Bussen anliegen (z.B. ein Scheibenwischer-Steuergerät, das seine Signale über einen CAN-Bus bekommt und an seine über einen lokalen LIN-Cluster angebundenen Aktoren weitergibt); deren Simulation vereinfacht sich durch die Bereitstellung von Gateway-Funktionalität durch ein Restbussimulations-Framework.

1.7 Rolle der Code-Generierung

Der Umfang der Kommunikationsmatrix und die dieser zu Grunde liegende Regularität führt dazu, dass eine manuelle Implementierung der Kommunikations-Strukturen zum Zwecke der Simulation teuer und fehleranfällig ist, so dass sich der Einsatz von Code-Generatoren hier aufdrängt, um eine konsistente Basis der durch die Kommunikationsmatrix definierten Strukturen zu schaffen [ZS10, KF09].

Eine dynamische Interpretation der Kommunikationsmatrix zur Laufzeit, d.h. die Ausführung des definierten zyklischen Verhaltens durch eine generische Laufzeit-Bibliothek, ist prinzipiell möglich, setzt aber entsprechende Hardware-Ressourcen für das System voraus, das die Restbussimulation ausführt, und schließt eine Portierung auf beschränktere Hardware-Plattformen kategorisch aus. In Fällen, wo dies akzeptabel ist, insbesondere im Falle von host-gebundenen Restbussimulationen (d.h. Restbussimulationen, die nicht autonom auf dedizierter Hardware, sondern unter der Kontrolle eines Entwicklungsrechners mit entsprechender Umgebung laufen), ist dies jedoch durchaus eine praktikable Lösung. Für eine dynamische Restbussimulation stellt sich bei diesem interpretativen Ansatz die Frage, wie die eigentlichen Simulationsaspekte, d.h. das nicht durch die Kommunikationsmatrix beschriebene Verhalten, durch den Benutzer implementiert werden können (siehe hierzu Abschnitt 2.5.3).

Kapitel 2

Architekturen für Restbussimulationen

In diesem Kapitel werden die grundsätzlichen Anforderungen an Architekturen für Restbussimulationen dargelegt und, in Abschnitt 2.6, mit einer Referenz-Architektur für Seriensteuergeräte im Gesamtfahrzeug kontrastiert.

2.1 Aufgaben der Restbussimulation

Eine Restbussimulation muss die folgenden unterschiedlichen Kernaufgaben erfüllen (siehe Abbildung 2.1):

- die Verarbeitung der Bus-Botschaften im Sinne der in den Datenbanken spezifizierten Parametern (Signalverarbeitung)
- die Ereignisverarbeitung, d.h. die Anbindung externer Ereignisse an die Zielplattform und die Weiterleitung generierter Ereignisse an die Zielplattform
- das Ausführen von benutzerdefinierten oder extern generierten Erweiterungen, die an die Ereignisverarbeitung gebunden sind
- die Beeinflussung des Sendeverhaltens von Teilen der simulierten Netzwerkteilnehmer in Reaktion auf Ereignisse oder Fehlerfälle

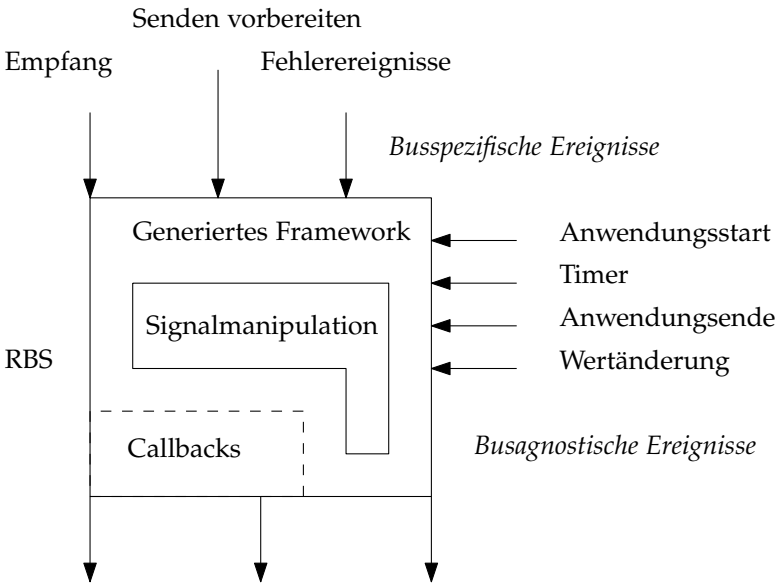

Abbildung 2.1: Restbussimulation, bestehend aus Signalverarbeitungs-Kernel, Plattform-Anbindung und benutzerdefinierter Callback-Funktionalität [Ber]

2.1.1 ECU-Simulation

Jede für die Restbussimulation relevante ECU muss als Software-Komponente in der generierten Anwendung abgebildet sein. Bei ECUs, die auf mehreren Bussen aktiv sein können, empfiehlt sich, insbesondere im Falle von unterschiedlichen physikalischen Bussen, die Simulation der ECU aufzuteilen in einzelne Komponenten, die jeweils die Anbindung an den jeweiligen Bus realisiert (siehe Abbildung 2.2). Interne Kommunikation innerhalb der ECU ist problemlos möglich, da die Restbussimulation und damit alle logischen Knoten der ECU innerhalb eines Addressraumes ausgeführt werden. Diese Methode, ECUs wenn nötig in busspezifische Knoten zu separieren, hat den Vorteil, dass Gateways nicht als Sonderfälle behandelt werden müssen.

2.1.2 Signalverarbeitung

Die Beschreibungsdateien für die Kommunikationsmatrix definieren für Nachrichten, die über die unterstützten Bussysteme übertragen werden, Interpretations-Vorschriften, die es ermöglichen, aus dem auf Busebene als abstraktes Byte-Array vorliegenden Nachrichteninhalt Signal-Informationen zu extrahieren und zu diesen extrahierten Signal-Werten eine physikalische Umrechnungs-Vorschrift

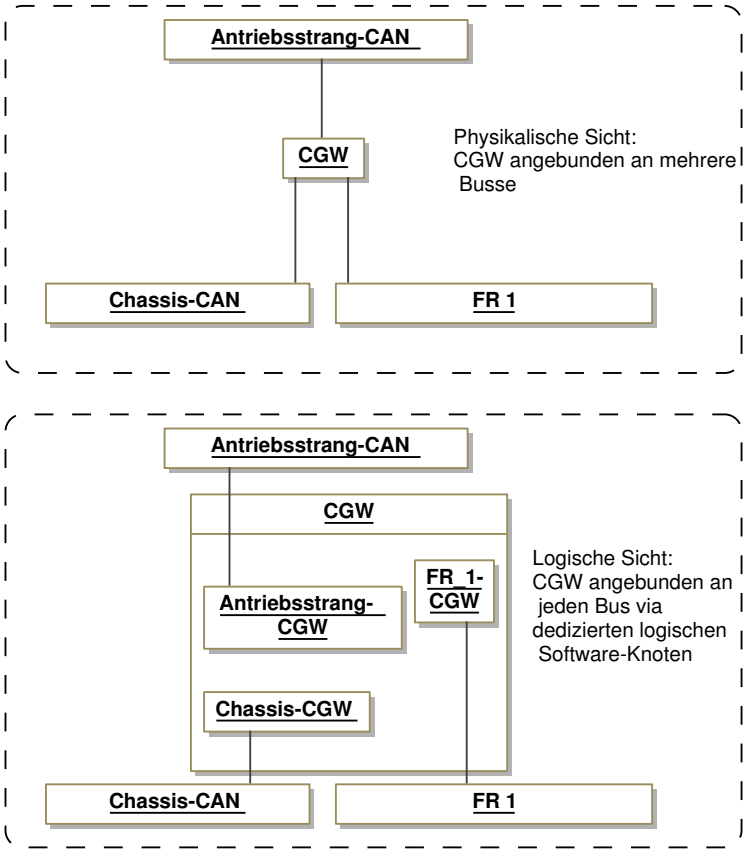

Abbildung 2.2: ECU-Sicht und logische Abbildung in Software-Komponenten am Beispiel einer Gateway-ECU [Ber]

zu definieren.

Es gibt einige Aspekte, die diese Funktionalität in mancher Hinsicht komplexer macht als vergleichbare Beschreibungsformate für Binärformate, wie sie z.B. in Middleware-Systemen in der PC-Welt vorkommen (Beispiele für diese Beschreibungen sind z.B. CORBA, Protocol Buffers oder BSON):

Nachrichten mit Signalen unterschiedlicher Byte-Reihenfolge Innerhalb einer Nachricht dürfen Signale mit unterschiedlicher Endianness, d.h. Byte-Reihenfolge, vorkommen. Die zwei von allen Beschreibungsformaten erlaubten Varianten sind hier die Intel-Reihenfolge (genannt *little endian*;

das Byte, das das höchstwertige Bit enthält, belegt die höhere Speicher-Adresse) und die Motorola-Reihenfolge (genannt *big endian*; das Byte, das das höchstwertige Bit enthält, belegt die niedrigste Speicher-Adresse). Sollen Signale arithmetisch verarbeitet werden, so müssen sie in das Format konvertiert werden, das vom verwendeten Prozessor verwendet wird, d.h. Signale im Intel-Format müssen bei Verwendung einer Prozessor-Architektur, die das Motorola-Format unterstützt, in dieses konvertiert werden. Signale, die im Motorola-Format vorliegen, müssen entsprechend auf Plattformen mit Intel-Format in dieses konvertiert werden.

Signale, die Byte-Grenzen verletzen Es gibt Signale, die nicht an Byte-Grenzen orientiert sind, d.h. die Teile eines Bytes und Teile eines angrenzenden Bytes belegen.

Signale beliebiger Länge Bei Signalen, deren Länge nicht genau 8, 16, 32 oder 64 Bit beträgt, ergibt sich die Problematik, dass das Signal in der Restbussimulation durch einen C-Typ repräsentiert werden muss, der einem dieser Längen entspricht (der kleinstmögliche dieser Typen bezogen auf die Signal-Länge), d.h. das Signal wird durch eine Variable verwaltet, von der nur ein bestimmter Bereich verwendet werden darf.

Die häufigsten in dieser Kategorie auftretenden Fälle sind Signale, die kleiner sind als ein Byte. Dies ist für Signale, deren C-Typ als `unsigned` definiert ist, kein Problem (die höherwertigen Bits, die nicht Teil des Signals sind, können auf 0 gesetzt werden). Arithmetische Operationen funktionieren wie erwartet. Bei Signalen, deren C-Typ als `signed` definiert ist, ergibt sich das Problem, dass hier der Rohwert im Zweier-Komplement vorliegt, dieser aber nicht einfach durch Nullsetzen der höheren Bit-Stellen in den (größeren) C-Typ übernommen werden kann. Stattdessen muss durchgeführt werden, was als sogenannte Sign-Extension bekannt ist, bzw. diese muss umgangen werden.

Ein so extrahiertes Signal liegt nach diesen Nachbehandlungen als Rohwert vor. Für manche Arten von Signalen, z.B. einfache Flags oder Enumerationen, die keine skalare Größe, sondern bestimmte diskrete Zustände repräsentieren, ist dies ausreichend, d.h. sie sind semantisch direkt verwertbar. Für skalare Werte muss u.U. zusätzlich eine Umrechnung vom Rohwert in den physikalischen Wert erfolgen, falls dies in der Beschreibungsdatei angegeben wird. Üblicherweise ist dies notwendig, wenn bestimmte Sensorwerte auf Seite des Senders exakt so in die Nachricht gepackt werden, wie sie der Sensor liefert (z.B. 12 Bit für viele

Standard-A/D-Wandler oder Winkelgeber). In diesem Fall muss eine Umrechnung von diesem Rohwert in eine physikalische Größe erfolgen. Üblichstes Verfahren ist eine lineare Umrechnung, d.h. eine Umrechnung mittels einer linearen Funktion, die eine Skalierung des Rohwertes, gefolgt von einer zusätzlichen Verschiebung, realisiert. Einzelne Beschreibungsformate erlauben es, unterschiedliche lineare Umrechnungen für verschiedene Wertebereiche des Rohwertes zu definieren oder ein explizites Mapping von Werten durchzuführen, d.h. bestimmte diskrete Werte des Rohwertes auf andere diskrete Werte abzubilden (die letztere Variante kommt z.B. im Rahmen der Netzwerk-Management-Funktionalität häufig zum Einsatz).

Sämtliche in diesem Abschnitt beschriebenen Konversionen und Umrechnungen müssen selbstverständlich auch in der umgekehrten Richtung erfolgen, d.h. es muss auch der Weg vom physikalischen Wert zum Rohwert und von diesem zum in die Botschaft integrierten Signalwert realisiert werden.

Als zusätzliche Anforderung bei diesem Aspekt ergibt sich die Notwendigkeit, eine gewisse Transaktionalität zu schaffen, so dass Lese- und Schreibvorgänge auf Nachrichtenpuffern bei Verwendung mehrerer Threads keine unerwünschten Effekte erzeugen. Ein Beispiel ist die Vermeidung des so genannten Tearing, das auftritt, wenn ein Thread ein sich über mehrere Byte erstreckendes Signal in einen Puffer integriert, und ein anderer auf diesen Puffer lesend zugreift, während erst Teile des Signals in den Puffer geschrieben wurden. Die Einbindung dieser Synchronisations-Mechanismen muss möglichst geringen Einfluss auf die Performance haben; hier bietet sich z.B. die Verwendung von Read-Write-Locks an, die mehreren lesenden Threads ermöglichen, gleichzeitig auf ein geschütztes Objekt zuzugreifen, aber nur den Zugriff durch einen einzelnen schreibenden Thread erlaubt (der wiederum nur Zugriff erhält, wenn kein lesender Thread zugreift) [Ker10]. Es geht hier nicht nur um den Aspekt der Synchronisation im temporalen Sinne, d.h. die Garantie des exklusiven Zugriffs auf eine Ressource für eine bestimmte Zeitspanne, sondern auch um die Speicher-Synchronisation (durch Einrichtung von Memory-Barrieren), die je nach verwendetem Betriebssystem und Threading-Modell nur an bestimmten Synchronisationspunkten erfolgt; ohne diese Barrieren würde das C-Speicher-Modell erlauben, dass der Compiler den Zugriff auf von mehreren Threads verwendete Variablen so optimiert (z.B. durch die zeitweise Verwendung eines Registers für eine eigentlich speicherresidente Variable), dass nicht garantiert werden kann, dass Änderungen an einer Variablen aus einem Thread jemals für andere Threads sichtbar werden. [1]

[1] Unter `pthread`-Implementierungen dienen die Methoden zur Steuerung des exklusiven Zugriffs gleichzeitig auch als Memory-Barrieren, so dass in der Regel hier kein zusätzlicher Aufwand entsteht.

2.1.3 Ereignisorientiertes Anwendungsmodell

Über die Qualität einer Restbussimulation entscheidet deren Fähigkeit, eingehende Ereignisse zu verarbeiten und ausgehende Ereignisse zu generieren. Jegliche Bewertung der Qualität muss diesen Aspekt mit hinreichender Gewichtung miteinbeziehen, und das Verarbeitungs-Modell für Ereignisse muss eine zentrale Rolle in der Architektur der Restbussimulation spielen. Für eingehende Ereignisse gibt es hierbei keine elementaren Unterschiede zu üblichen Server-Anwendungen (allerdings mit deutlich kleineren Paketgrößen und deutlich höherer Ereignisdichte). Für die ausgehenden Ereignisse gibt es jedoch den zentralen Aspekt der Zeit: für die Restbussimulation ist es in der überwiegenden Zahl der Anwendungsfälle nicht zielführend, Antworten auf Ereignisse möglichst schnell zu senden, sondern diese müssen in aller Regel zu definierten Zeitpunkten gesendet werden, wobei die Mechanismen und Anforderungen an das temporale Verhalten stark von den Rahmenbedingungen der Restbussimulation abhängen, insbesondere den eingesetzten Bussystemen und dem Lastprofil des konkreten Anwendungsfalles.

Im Folgenden werden die grundsätzlichen Ereignisse beschrieben, die für die Restbussimulation zugänglich sein müssen.

2.1.4 Busspezifische Ereignisse

Die folgenden Ereignisse sind in ihrer Schnittstelle und teilweise in ihrer Semantik abhängig vom Bus-Typ, über die das Ereignis ausgelöst wird. In der laufenden Restbussimulation können sie einem konkreten Bus-Kanal zugeordnet werden.

Botschaftsempfang Dieses Ereignis wird ausgelöst, wenn der Bus-Controller eine gültige Nachricht empfangen hat und dies dem Betriebssystem bzw. bei Systemen ohne Betriebssystem direkt der Anwendung über einen Interrupt signalisiert. Der Inhalt der Nachricht wird in den Speicherbereich der Anwendung übernommen und u.U. eine vom Benutzer definierte Funktion aufgerufen, die eine Referenz auf diesen Nachrichtenpuffer als Parameter erhält.

Botschaftsübermittlung Häufig gibt es Nachrichten, in denen bestimmte Felder eine besondere Semantik besitzen (Beispiel: Prüfsumme oder fortlaufender Nachrichtenzähler); diese Felder müssen nicht nur bei Änderungen an Signalwerten neu gesetzt werden, sondern vor jedem Senden der Nachricht (unabhängig von vorhandenen Änderungen an regulären Signalen derselben Nachricht). Aus diesem Grund muss auch das bevorstehende Senden einer Nachricht als Ereignis zur Erweiterung durch den Benutzer

zur Verfügung stehen (wiederum als Callback-Funktion mit einer Referenz auf die zu sendende Nachricht als Parameter). Dies ermöglicht es, unmittelbar vor dem Senden eine anwendungsspezifische Prüfsumme zu berechnen oder einen Nachrichtenzähler zu erhöhen.

Fehler auf Busebene Fehler auf physikalischer Ebene werden direkt vom Controller (u.U. sogar direkt vom Transceiver) behandelt; dies ist notwendig, da je nach Busprotokoll eine Maskierung einzelner Fehler durch den Controller erfolgen muss (z.B. durch das automatische erneute Senden fehlerhaft übertragener Nachrichten). Unter CAN gibt es definierte Controller-Zustände [ZS10], die nach bestimmten Häufungen von Fehlern eingenommen werden und die für die Restbussimulation relevant sind und an diese weitergeleitet werden. Typische Reaktionen der Anwendung auf solche Ereignisse sind z.B. der Übergang in einen benutzerdefinierten sicheren Zustand beim Übergang des Controllers in einen Fehlerzustand und das Wiederaufnehmen der regulären Aktivität beim Wiedereintritt des Controllers in einen aktiven Zustand sowie das Loggen des Ereignisses.

Eine explizite Einsicht in jedes Fehlerereignis (oder das anwendungsgesteuerte Erzeugen solcher Fehler) auf Busebene ist für solche Anwendungsfälle interessant, die sich mit der Analyse der Bus-Topologie und der Prüfung des Verhaltens von Controllern befassen; dies sind jedoch keine Anwendungsfälle für die Restbussimulation, sondern für spezielle Fault-Injection-Werkzeuge, die an bestimmte Hardware mit Bus-Controllern gebunden sind, die einen niedrigeren Abstraktionsgrad als die Standard-Bus-Controller aufweisen müssen, bzw. die an spezielle Transceiver gebunden sind, die dynamisch neu konfiguriert werden können, ein bestimmtes Fehlverhalten zu zeigen.

2.1.5 Busagnostische Ereignisse

Die folgenden Ereignisse werden von der Restbussimulation selbst erzeugt oder deren Erzeugung wird durch die Restbussimulation veranlasst, um mit benutzerdefinierten Callback-Funktionen darauf reagieren zu können. Sie sind unabhängig von der Anbindung der Restbussimulation an konkrete Bus-Kanäle.

Anwendungsstart Beim Start der Restbussimulation müssen Parameter für Bus-Kanäle gesetzt werden, die Bedatung [2] der Datenpuffer gemäß den Informationen in den verwendeten Datenbasen oder ggf. durch vorgegebe-

[2] Terminus für die Vorbelegung von Datenwerten

ne Default-Werte durchgeführt und (im Entwicklungs-Modus) Server für Kontroll- und Tracer-Kanäle initialisiert werden.

Anwendungsende Beim Beenden der Restbussimulation müssen die Parameter für Bus-Kanäle zurückgesetzt werden bzw. diese deaktiviert werden.

Zeitereignisse Die Restbussimulation muss für relative oder absolute Zeitpunkte benutzerdefinierte Funktionen ausführen können. Verwendet werden kann dies z.B. zur Veränderung von Signalwerten gemäß einer gewünschten zeitabhängigen Kennlinie. Andere Anwendungsfälle sind das Sampling von analogen oder digitalen Eingangssignalen durch eine benutzerdefinierte periodische Funktion.

Signalwert-Änderungen Signale werden zwar als Teil von Botschaften zwischen Teilnehmern kommuniziert und sind damit prinzipiell busspezifisch; sie sind innerhalb einer Restbussimulation jedoch als eigenständige Elemente manipulierbar und haben aus Benutzersicht keine direkte Assoziation mit dem verwendeten Kommunikationsmechanismus (diese Abstraktion ist ja gerade eine der Hauptaufgaben des Restbussimulations-API). Änderungen an Signalwerten müssen im Hinblick auf definierte Wertebereiche überwacht werden, wobei es weitgehend anwendungsabhängig ist, ob unzulässige Änderungen unterbunden werden, auf die zulässigen Grenzwerte abgebildet werden oder Warnungen auslösen.

Prozessvariablen-Änderungen Um während früher Entwicklungsphasen externe Größen (Sensorwerte etc.) zu simulieren oder um Vorgabewerte an eine laufende Restbussimulation weiterzuleiten werden in der Regel so genannte Prozessvariable verwendet. Sie ermöglichen einen kontrollierten Zugriff auf den gekapselten Datenwert und propagieren diesen bei Änderungen an abhängige Funktionen und ermöglichen so eine Variante von Dataflow-Programmierung, d.h. das systematische Anbinden von Ereignisbehandlungsroutinen und das automatische Berechnen abgeleiteter Größen bei Änderungen. Häufig motiviert sich die Dataflow-Semantik auch aus den graphischen Darstellungen, die viele Simulations-Umgebungen bieten (vermutlich bekanntestes Beispiel hierfür sind Matlab-Simulink-Blöcke, bei denen der Datenfluss als Verbindung der Ausgangs-Ports von Datenquellen mit den Eingangs-Ports von Datensenken modelliert wird).

Dieser Mechanismus unterscheidet sich hinsichtlich der zuvor genannten Manipulation von Signalwerten dadurch, dass in diesem Fall die Variablen nicht in den Datenbasen definiert sein müssen, sondern vom Benutzer

Abbildung 2.3: Hardware-Plattform *ISI CANyon* [Ber]

explizit eingeführt werden. Sie stellen in der Regel Aspekte dar, die nicht Element der durch die Datenbasen modellierten Architektur sind, sondern speziell für die Restbussimulation in einem bestimmten Umfeld (z.B. am Prüfstand) benötigte Parameter, z.B. festgelegte Laufzeit, Kennlinienverläufe, Diagnose- und Debug-Informationen.

2.2 CAN-Kommunikation unter Linux

Da die Referenz-Implementierung für ein Linux-System auf PowerPC-Basis erfolgt (das *ISI CANyon* von Berger Elektronik, konzipiert von Bastian Hitzler in [Hit10]; siehe Abbildung 2.3), werden im Folgenden einige Aspekte von Linux als Softwareplattform erläutert, mit dem Fokus auf die Untertützung des vorgestellten ereignisorientierten Anwendungsmodells. Für Leser, die bereits auf Erfahrungen in der Linux-Systemprogrammierung zurückblicken dürfen, ist dennoch Abschnitt 2.2.4 relevant, da dieser auf die Einbindung von CAN eingeht, was ein eher wenig bekannter Teil des Linux-Kernels sein dürfte.

2.2.1 Klassifikation von Anwendungen

In der Systemprogrammierung gibt es zwei unterschiedliche Kategorien, in die Tasks [3] hinsichtlich ihres Ressourcenbedarfs eingeordnet werden: I/O-gebundene Tasks und CPU-gebundene Tasks.

I/O-gebundene Tasks verbringen den wesentlichen Teil ihrer Zeit in Wartezuständen, wobei der Begriff „Warten“ in diesem Zusammenhang entweder das Warten auf die Gewährung eines exklusiven Nutzungsrechts für eine bestimmte Ressource beschreibt oder das Warten auf das Auftreten von Ereignissen im Zusammenhang mit einer bestimmten Ressource (z.B. auf Eingabe- und Ausgabe-Ereignisse), wobei es zu großen Teilen betriebssystemabhängig ist, welche Ereignisse zur Verfügung stehen. Die bei weitem am häufigsten verarbeiteten Ereignisse sind die Ein- und Ausgabe von Daten über Netzwerk- oder Dateisystem-Schnittstellen.

CPU-gebundene Tasks verbringen den wesentlichen Teil ihrer Zeit mit Berechnungen, d.h. der aktiven Verwendung eines Prozessorkerns. Beispiele sind algorithmisch aufwändige Datentransformationen und Berechnungen.

Im Hinblick auf Linux-Systeme ist eine ähnliche, aber nicht völlig identische Klassifikation in Syscall[4]-lastige Tasks und User-Space-lastige Tasks möglich. Wenn ein Programm einen Syscall ausführt, werden immer zwei Kontextwechsel durchgeführt (nach dem Aufruf aus dem User-Space wird in den Kernel-Kontext gewechselt und vor der Rückkehr wird vom Kernel-Kontext in den User-Space gewechselt), der zudem weitreichendere Konsequenzen hat und höhere Kosten verursacht als ein Kontextwechsel zwischen unterschiedlichen User-Space-Tasks. Diese höheren Kosten für Syscalls im Gegensatz zu einfachen Funktionsaufrufen innerhalb einer Anwendung liegen unter anderem an den unterschiedlichen Adressräumen, in denen der Kernel und der Anwendungsprozess laufen (hier sind immer Puffer-Kopien notwendig statt wie beim Aufruf innerhalb der Anwendung nur Referenzen zu übergeben), sowie an der Notwendigkeit des Kernels, eine viel gezieltere Überprüfung von Parametern durchzuführen, um bestimmten

[3] Im Folgenden wird der Begriff Task als Überbegriff für Prozesse und Threads verwendet. Es ist betriebssystemabhängig, inwieweit sich Threads und Prozesse unterscheiden. Eine generelle Unterscheidung ist dadurch gegeben, dass Prozesse in einem logisch voneinander unabhängigen Adressraum ablaufen, während Threads immer den Adressraum des Elternprozesses verwenden. Somit sind Prozesse gegenseitig besser isoliert, mit den damit verbundenen Vorteilen und Einschränkungen.

[4] Ein Syscall bezeichnet den Aufruf einer im Kernel definierten Routine, die im geschützten Speicherbereich des Kernels abläuft; zur Abgrenzung gegen den allgemeinen Begriff „Systemaufruf“, der sich u.U. auch auf Aufrufe von in vom Betriebssystem bereitgestellten Bibliotheken vorhandene Funktionen ausdehnen lässt, hat der Begriff „Syscall“ eine deutlich präzisere Definition.

Sicherheitsaspekten gerecht zu werden, die zwar nur im traditionellen Mehrbenutzerbetrieb essentiell sind, die aber nicht einfach deaktiviert werden können.

In der überwiegenden Mehrheit der Anwendungsfälle für konkrete Restbussimulationen dominieren die Anforderungen an die Ereignisverarbeitung die Architektur; algorithmisch aufwändige Restbussimulationen (man denke an das Prototyping von ESP-Steuergeräten) sind durchaus möglich, werden jedoch in der Praxis im Prototypen-Stadium meist innerhalb einer Umgebung zur numerischen Simulation (z.B. Matlab) entwickelt, da in diesem Fall das algorithmische Prototyping (d.h. die Entwicklung und Optimierung des Algorithmus) den Entwicklungsaufwand dominiert.

Eine Integration algorithmisch aufwändiger Anwendungen in ein ereignisorientiertes System ist meist durch eine Diskretisierung des Algorithmus möglich. Der umgekehrte Fall, die Integration von hinreichend präzisen Ereignisbehandlungs-Funktionen in ein System mit blockierenden Berechnungen, ist dagegen nicht möglich (bzw. wiederum nur durch eine Diskretisierung der Berechnung), so dass eine ereignisorientierte Architektur als Voraussetzung für Restbussimulationen gelten kann.

2.2.2 I/O-Multiplexing

Das ereignisorientierte Anwendungsmodell führt dazu, dass die Anwendung einen wesentlichen Teil ihrer Laufzeit mit dem Warten auf Ereignisse verbringt. Um diese Wartezustände sinnvoll handzuhaben, d.h. die Ausführung von Funktionen auch während des Wartens zu ermöglichen, gibt es zwei unterschiedliche Ansätze, die kombiniert werden müssen, um ein effizientes Warten auch bei einer größeren Menge an potentiellen Ereignissen zu ermöglichen.

Einerseits muss die Anwendung in verschiedene Threads partitioniert werden, so dass die Blockierung des Kontrollflusses durch das Warten in einem Thread die lauffähigen Teile der Anwendung nicht beeinträchtigt. In der Praxis ist es jedoch nicht möglich, jeden Ereignistyp durch einen eigenen Thread vom Rest der Anwendung zu entkoppeln, da dies einerseits Arbeitsspeicher für die Thread-Verwaltung benötigt (ein eigener Stack pro Thread sowie diverse Verwaltungsinformationen) und andererseits CPU-Zeit verbraucht und interne Cache-Invalidierungen und Memory-Bus-Last durch Kontext-Wechsel verursacht [Ker10]. Bei neueren Linux-Systemen ist zwar ein sogenannter O(1)-Scheduler vorhanden, bei dem die Zeit zum Kontext-Wechsel zwischen zwei Tasks nicht von der Gesamtzahl der Tasks abhängig ist, sondern konstant [Hal11]. Die Gesamtzeit für das Scheduling zwischen allen Tasks ist jedoch weiterhin abhängig von der Ge-

samtzahl der Tasks, gemäß der Formel $t_{CPU} = \sum_n t_n + \sum_n C_{cs} = \sum_n t_n + nC_{cs}$, mit t_{CPU} als Zeit für einen kompletten Scheduling-Zyklus für alle als gleich priorisiert angenommenen Tasks, n als Anzahl der Tasks, t_n als Zeit für Task n im betrachteten Zyklus und C_{cs} als konstante Zeit für einen einzelnen Kontext-Wechsel. [5]

Es ist also darüber hinaus notwendig, innerhalb eines einzelnen Threads nicht nur auf ein bestimmtes Ereignis warten zu können, sondern auf eine Menge an Ereignissen (um so die Anzahl der für die Ereignisbehandlung dedizierten Threads drastisch zu verringern). Der Standard-Mechanismus dafür ist unter POSIX-Systemen die Verwendung der `select`-Funktion[6], die auf Ereignisse einer Menge von Dateideskriptoren wartet und das Warten unterbricht, wenn für mindestens einen dieser Dateideskriptoren Ereignisse signalisiert werden. Hierbei ist zu erwähnen, dass dies selbstverständlich kein aktives Warten und Abfragen der Menge an Deskriptoren im User-Space durch die Anwendung bewirkt (d.h. kein Polling), sondern die Abgabe des Kontrollflusses der Anwendung (Descheduling des Threads) und eine durch den Kernel initiierte Wiederaufnahme desselben im Falle von Ereignissen auf den überwachten Deskriptoren. Die unterschiedlichen Modelle sind in Abbildung 2.4, Abbildung 2.5, Abbildung 2.6 und Abbildung 2.7 dargestellt.

Prinzipiell ist es möglich, sämtliche an einen Dateideskriptor gebundenen Ereignisse so in einem einzelnen Thread zu behandeln. Dies führt jedoch leicht zu einer softwaretechnisch unschönen Konglomeration von eigentlich unabhängigen Aspekten. Dies mag vor dem Hintergrund der Code-Generierung nicht dramatisch erscheinen (der Benutzer kommt mit dem intern verwendeten Mechanismus nicht direkt in Kontakt), ist allerdings für die Wartbarkeit und Erweiterungsfähigkeit der Code-Generatoren nachteilig. Insbesondere gibt es ohnehin Ereignisse, die nicht auf Dateideskriptoren abgebildet werden können, so dass zwangsläufig alternative Methoden hinzugezogen werden müssen. Ein Vorteil der Einzel-Thread-Lösung wäre allerdings, dass die Abbildung auf ein microcontrollernahes (durch Interrupts getriebenes) Verarbeitungsmodell oder eine Echtzeit-Variante von Linux so einfacher möglich ist.

Eine Alternative zur `select`-Funktion ist (sofern auf die POSIX-Kompatibilität verzichtet werden kann und eine Linux-Lösung akzeptabel ist) die Verwendung des `epoll`-API, das bei einer größeren Menge an zu überwachenden Dateideskriptoren effizienter ist als `select`-Aufrufe [Keg].

[5] Eine ausführliche Beschreibung der Thematik ist zu finden unter [Keg].

[6] Das Socket-API von Windows unterstützt zwar eine Variante von `select`, jedoch ist diese auch nur mit Sockets und nicht mit beliebigen Dateideskriptoren verwendbar.

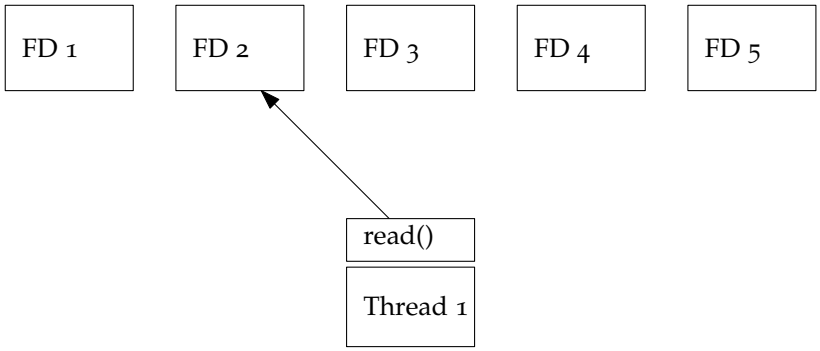

Abbildung 2.4: Ein einzelner Thread; `read()`-Call suspendiert den Thread, bis der Kernel ein Ereignis auf dem überwachten Dateideskriptor signalisiert.

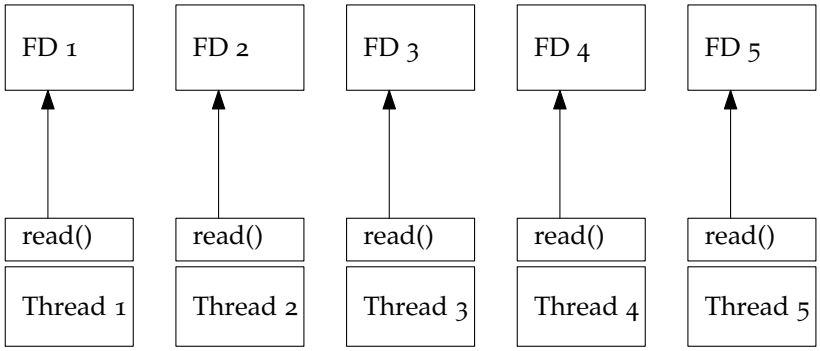

Abbildung 2.5: Mehrere Threads; `read()`-Call suspendiert den aufrufenden Thread, bis der Kernel ein Ereignis auf dem überwachten Dateideskriptor signalisiert. [Ber]

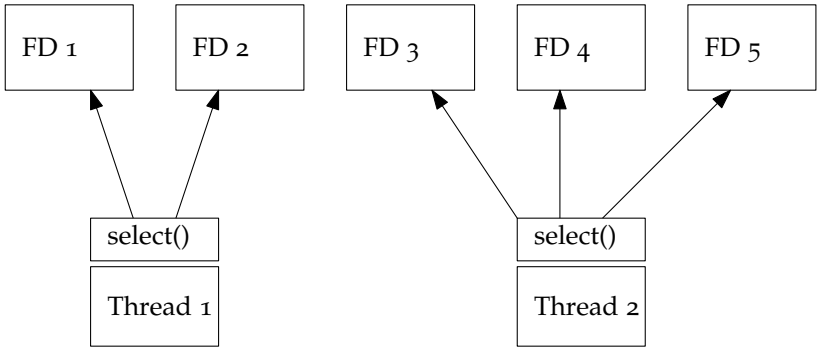

Abbildung 2.6: Mehrere Threads; `select()`-Call suspendiert den aufrufenden Thread, bis der Kernel ein Ereignis auf mindestens einem der überwachten Deskriptoren signalisiert. [Ber]

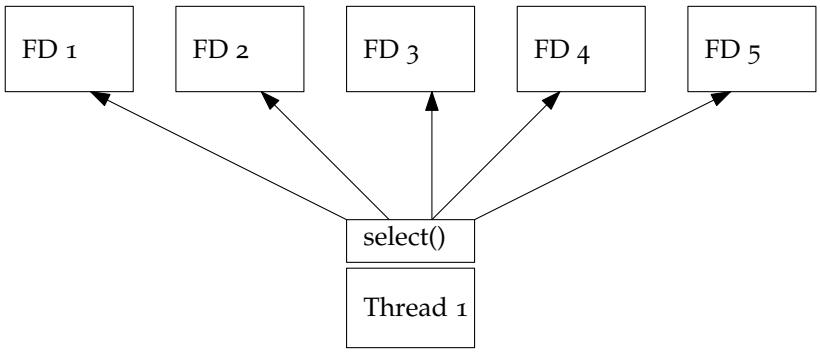

Abbildung 2.7: Ein einzelner Thread; `select()`-Call suspendiert den Thread, bis der Kernel ein Ereignis auf mindestens einem der überwachten Deskriptoren signalisiert. [Ber]

2.2.3 Präzises Scheduling im User-Space

Um mit hinreichender Genauigkeit zu bestimmten Zeitpunkten Botschaften senden zu können, muss ein Zeitgeber mit einer Auflösung im Bereich von etwa 100 Mikrosekunden verfügbar sein.[7] Dies motiviert sich aus der Tatsache, dass bei den meisten Beschreibungsformaten für die Buskommunikation die Zykluszeit von Botschaften mit einer Millisekunde Genauigkeit angegeben wird. In der Regel werden bei sicherheitskritischen Anforderungen (bei denen eine potentielle Totzeit von einer Millisekunde nicht akzeptabel wäre) die Botschaften ausserhalb eines eventuell vorhandenen Zyklus unmittelbar als Botschaften hoher Priorität gesendet (zumindest bei CAN-Kommunikation, wo dies möglich ist, da kein Scheduling auf Protokollebene einzuhalten ist). Bei Verwendung von CAN nimmt die reine Übertragungszeit für eine Nachricht bereits mehr als 100 Mikrosekunden in Anspruch [Rei08]. Die `clock_nanosleep`-Funktion [8] bewirkt unter Linux das Warten auf den Ablauf des gegebenen Intervalls oder auf das Erreichen eines absoluten Zeitpunktes.

In [Ker10] werden mehrere Linux-spezifische Ansätze vorgestellt, wie die Realisierung von Funktionsaufrufen zu präzisen Zeitpunkten realisiert werden kann, darunter der `timerfd`-Ansatz, bei dem spezielle virtuelle Dateideskriptoren als Zeitgeber erzeugt werden können und somit in eine reguläre `select`-Schleife

[7] Dies betrifft das Scheduling von CAN- und LIN-Botschaften; unter Flexray erfolgt das eigentliche Senden durch den dedizierten Flexray-Controller, da die Präzisionsanforderungen hier ungleich höher sind.

[8] Die selbstverständlich, entgegen dem Anschein, der durch den Funktionsnamen erweckt wird, keine Nanosekunden-Auflösung hat, sondern eine systemspezifische Auflösung.

eingebunden werden können und somit auf Ereignisse des Zeitgebers und I/O-Ereignisse mit dem gleichen Mechanismus reagiert werden kann.

Würde die Restbussimulation als Kernel-Modul laufen (siehe Abschnitt 2.5.2), so könnte das Timing durch die Kernel-Timer erfolgen, die sich durch höchstmögliche Präzision (für die verwendete Plattform) und einfache Programmier-Schnittstelle (Trigger-Zeitpunkt mit auszuführender Callback-Funktion) auszeichnen. Der verwendete Timer-Wheel-Algorithmus zur Realisierung des Scheduling der Kernel-Timer, wie sie im Kernel 2.6 realisiert sind, ist in [GN06] beschrieben.

2.2.4 SocketCAN

Das verwendete Linux-System unterstützt das CAN-Protokoll bzw. CAN-Hardware mittels der ursprünglich von der Volkswagen AG entwickelten so genannten SocketCAN-Treiber[9], die seit einiger Zeit Teil des Mainline-Kernels sind. Das Besondere an diesen ist, dass sie auf dem gewöhnlichen Netzwerk-Stack von Linux aufbauen und CAN als neue Protokollfamilie einführen. Technisch und konzeptuell hat CAN keinerlei Überschneidungen mit den üblichen von diesem Stack unterstützten Protokollen (unterschiedliche Bus-Physik, unterschiedliche Arbitrierung); Motivation für die Verwendung des Netzwerk-Stacks zur Anbindung von CAN-Controllern war ausschließlich die Nutzung der bestehenden Infrastruktur wie Puffer-Verwaltungs-Datenstrukturen und -Funktionen auf Kernel-Seite und die Nutzung von bestehenden, bewährten Datei- und Socket-APIs im User-Space. Durch die Integration in den Netzwerk-Stack ist es möglich, das POSIX-Socket-API zum Senden und Empfangen von CAN-Botschaften zu verwenden; die Einführung eines neuen API speziell für CAN wurde so vermieden und Anwendungsentwickler können bekannte und bewährte I/O-Muster für die CAN-Kommunikation verwenden.

Die grundsätzliche Funktionsweise besteht darin, dass von der CAN-Hardware (d.h. den angebundenen CAN-Controllern) abstrahiert wird, indem jeder vorhandene CAN-Kanal als eigenes Netzwerk-Interface angeboten wird, auf dem die vom Netzwerk-Stack vorgegebenen Operationen garantiert werden [Har]. Anwendungen instantiieren mittels des Socket-API logische bidirektionale Verbindungen zu diesen Netzwerk-Interfaces, wobei der Kernel als Multiplexer agiert, indem er mehreren Anwendungen erlaubt, dasselbe Gerät zu verwenden. Die Koordination des Zugriffs erfolgt hierbei durch den Kernel und muss nicht von den Anwendungen implementiert werden. Diese zwangsläufige Delegation der Aufgaben an den Kernel hat subtile Implikationen, auf die in Abschnitt 2.4

[9] Die ursprüngliche Bezeichnung für dieses API lautete *Low Level CAN Framework* (LLCF).

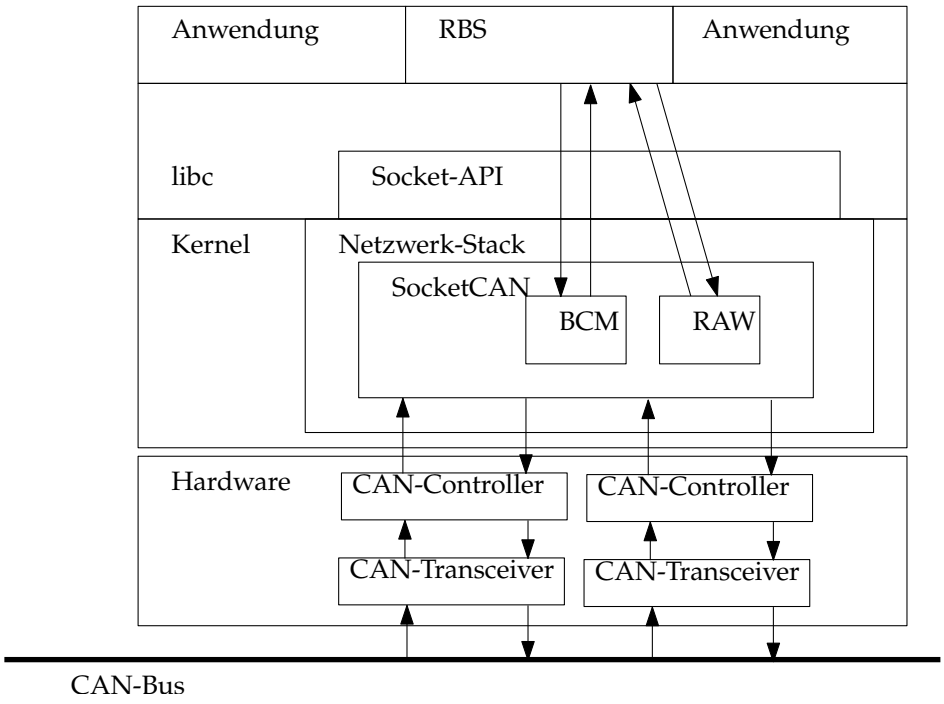

Abbildung 2.8: Abstraktionsschichten bei Verwendung von SocketCAN [Ber]

eingegangen wird.

SocketCAN bietet, neben den in dieser Arbeit nicht betrachteten Transportprotokollen, grundsätzlich zwei Arten von CAN-Sockets, RAW-Sockets und BCM-Sockets, die im Folgenden beschrieben werden und in Tabelle 2.1 gegenübergestellt werden, wobei beide Arten problemlos gemeinsam verwendet werden, sowohl innerhalb einer Anwendung als auch in unterschiedlichen Prozessen [Har].

Abbildung 2.8 zeigt schematisch die Kommunikations-Pfade zwischen Anwendung, Kernel und Hardware.

2.2.5 RAW-Sockets

Die RAW-Sockets entsprechen von der Verwendungsweise her regulären Sockets oder Dateideskriptoren: sie erlauben das Senden einer CAN-Botschaft mit den gewohnten `send` bzw. `write` Systemfunktionen und das Empfangen einer einzelnen CAN-Botschaft mit den `recv` bzw. `read` Systemfunktionen; SocketCAN garantiert hierbei, dass Aufrufe jeweils eine vollständige Botschaft komplett

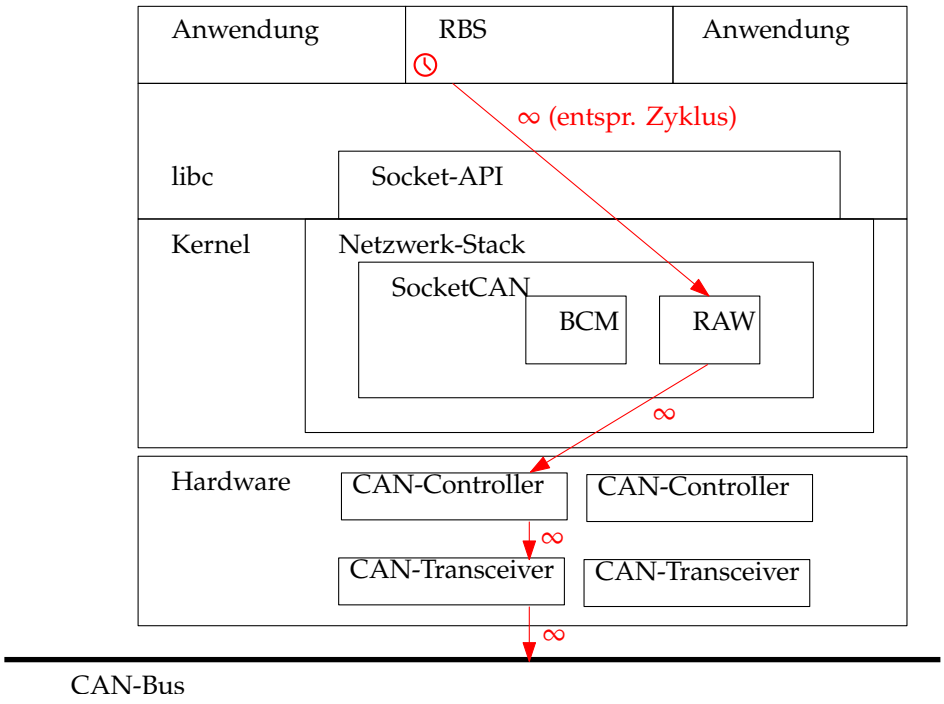

Abbildung 2.9: Datenübertragungs-Pfad bei Verwendung von RAW-Sockets. Bei jedem Senden muss die Userspace-Kernelspace-Grenze passiert werden, was Speicherkopien verursacht. Die Anwendung ist für das korrekte Timing verantwortlich. [Ber]

konsumieren bzw. liefern. Die RAW-Sockets sind echte Dateideskriptoren und können somit u. a. auch in `select` Systemaufrufen verwendet werden, um effizientes I/O-Multiplexing zu realisieren. RAW-Sockets bieten die Möglichkeit, Filter zu definieren, um den Empfang von Botschaften auf solche Botschaften beschränken, deren ID die Filterbedingungen erfüllen. Die Durchführung des Filterns im Kernel verhindert unnötige Kopien vom Kernel-Space in den User-Space. Je nach verwendetem CAN-Controller könnte die Filterung auch direkt auf der CAN-Hardware erfolgen; allerdings kann dies unter SocketCAN auf Grund des potentiellen intern stattfindenden Multiplexing von Nachrichten an verschiedene Sockets in verschiedenen Prozessen nicht durchgeführt werden [Har].

2.2.6 BCM-Sockets

Der Broadcast Manager (BCM) kann beschrieben werden als ein Management-API für zyklische CAN-Sendeaufträge: anstatt CAN-Botschaften direkt aus einer Anwendung zu senden, werden stattdessen dem BCM Sendeaufträge für Botschaften übermittelt, die zusätzlich zum Dateninhalt der Botschaften Informationen über Zykluszeiten enthalten. Der BCM (der im Kontext des SocketCAN-Moduls im Kernel läuft) übernimmt daraufhin die Verantwortung für das zyklische Senden der ihm übermittelten Botschaften; die Anwendung wird davon entlastet, insbesondere was das relativ aufwändig zu realisierende Zeitverhalten (Einhaltung der Zykluszeiten) betrifft. Zusätzlich werden unnötige Kopien zwischen User-Space und Kernel-Space vermieden, die sonst beim mehrmaligen Senden von Botschaften gleichen Inhalts nötig wären. Abbildung 2.10 stellt diesen Sachverhalt dar, während Abbildung 2.9 die Realisierung mit RAW-Sockets gegenüberstellt.

Soll der Dateninhalt einer vom BCM verwalteten Botschaft verändert werden, so kann dies als Änderungsauftrag über den BCM-Socket übermittelt werden; dabei kann angegeben werden, ob die veränderte Botschaft sofort gesendet werden und ein neuer Zyklus gestartet werden soll oder ob die veränderte Botschaft im Rahmen des nächsten regulären Zykluspunktes gesendet werden und der bestehende Zyklus beibehalten werden soll [Har]. Das Abbrechen eines Sendezyklus für eine CAN-Botschaft erfolgt analog.

Ein weiterer spezieller Anwendungsfall, den der BCM unterstützt, ist das Umschalten eines zyklischen Sendeauftrags von einem Intervall auf ein anderes Intervall nach einer definierten Anzahl von Durchläufen; relevant ist dies z.B. bei Nachrichten, die bei Einschaltvorgängen wichtig sind (z.B. zur Initialisierung anderer ECUs) und daher mit niedriger Zykluszeit gesendet werden sollen, im laufenden Betrieb aber eine höhere Zykluszeit ausreichend ist.

BCM-Sockets können auch zum Empfangen von Nachrichten verwendet werden. Hier ist zusätzlich zu den auch bei den RAW-Sockets vorhandenen ID-basierten Filtern eine inhaltsbasierte Optimierung möglich: der Socket kann so konfiguriert werden, dass eine Botschaft nur an die Anwendungsschicht weitergeleitet wird, wenn sich seit dem letzten Empfang dieser Botschaft nichts am Inhalt verändert hat. Hierbei ist durch Bitmasken konfigurierbar, welche Bereiche einer Botschaft für den Empfänger überhaupt relevant sind. Des Weiteren kann eine Zeitüberwachung erfolgen, indem für eine zu empfangende Nachricht ein Timeout-Wert t angegeben wird; wird nach dem ersten Empfang der Nachricht nicht nach spätestens t Zeiteinheiten die Nachricht erneut

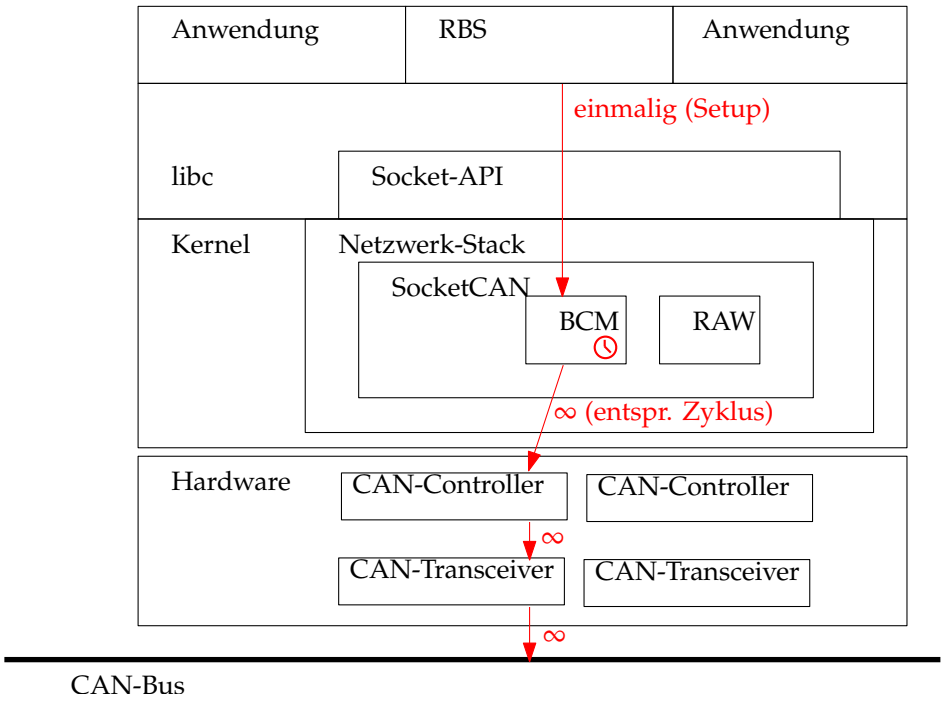

Abbildung 2.10: Datenübertragungs-Pfad bei Verwendung von BCM-Sockets. Die Anwendung muss die Daten für eine zyklisch zu sendende Botschaft nur einmalig (und bei Änderungen) an den Kernel übertragen. Das Timing erledigt das BCM-Kernelmodul. [Ber]

empfangen, erhält die Anwendung über den BCM-Socket eine entsprechende Benachrichtigung. Optional kann auf das Warten auf den Empfang mindestens einer Nachricht zum Starten der Überwachung verzichtet werden und unmittelbar mit der Überwachung begonnen werden.

Darüber hinaus wird die zeitbasierte Drosselung von an die Anwendung übermittelten Botschaften unterstützt, d.h. für eine Botschaft, deren Verarbeitung durch die Anwendung bei zu häufigen Änderungen nicht gewährleistet werden kann, kann explizit ein Minimal-Intervall angegeben werden, das zwischen zwei aufeinanderfolgenden Änderungen des Botschaftsinhaltes abgelaufen sein muss, um die Weiterleitung an die Anwendungsschicht zu veranlassen.

	RAW	BCM
Senden	×	×
Empfangen	×	×

	RAW	BCM
Empfangs-Multiplexing	×	×
Autonomes zyklisches Senden	-	×
Callbacks nach dem Empfangen	×	×
Callbacks vor dem Senden	×	×[10]
Error-Frames empfangen	×	×
Bus-Off-Benachrichtigungen empfangen	×	×
Timeout-Überwachnug	-	×
Filterung durch Botschafts-Identifier	×	×
Inhaltsbasierte Filterung durch Bitmasken	-	×
Umschaltung des Sende-Intervalls	-	×
Nutzung virtueller CAN-Bus	×	×
Drosselung von Botschaften	-	×

Tabelle 2.1: Vergleich RAW- und BCM-Sockets. Fehlende Funktionalität muss von der Anwendung realisiert werden.

2.2.7 Error-Frames und Bus-Off-Benachrichtigung

Für Anwendungen, die über Error-Frames auf dem Bus oder über die Abschaltung der Bus-Verbindung durch den CAN-Controller (als Reaktion auf Fehler) informiert werden müssen, gibt es bei beiden Socket-Arten die Möglichkeit, den Empfang entsprechender Botschaften, die standardmäßig nicht an die Anwendung weitergeleitet werden, zu aktivieren. Es handelt sich hierbei nicht um die eigentlichen Error-Frames, die auf dem Bus liegen, sondern um spezielle Nachrichten, die der SocketCAN-Treiber verwendet, um die Anwendung über diese Ereignisse zu informieren. Die Übermittlung erfolgt lokal zwischen Kernel und Anwendung und geht nicht über den CAN-Controller; es wird lediglich der bereits bestehende Socket zwischen Kernel und Anwendung genutzt, um für derartige Out-Of-Band-Informationen keinen orthogonalen Mechanismus verwenden zu müssen.

2.2.8 Virtueller CAN-Bus

SocketCAN ermöglicht es, virtuelle CAN-Kanäle zu definieren, mit denen die beschriebenen Socket-Arten ohne Änderungen verwendet werden können. Diese

[10] Jedoch nicht bei autonomen Sendeaufträgen.

simulieren die Kommunikation, ohne auf CAN-Hardware zuzugreifen bzw. ohne diese vorauszusetzen. Einfache Tests ohne Anspruch auf hohe Aussagekraft sind so trivial möglich [Har].

2.3 CAN-Kommunikation unter Windows

Unter Windows gibt es kein etabliertes CAN-Framework und keine einheitliche Abstraktion für Hardware unterschiedlicher Hersteller. Die XL Library von Vector Informatik, die sämtliche Hardware-Produktlinien dieses Herstellern mit sämtlichen Bussystemen unterstützt, kann als Basis für so genannte online Restbussimulationen verwendet werden, d.h. Restbussimulationen, die direkt auf einem Host-Rechner laufen statt auf einer dezidierten CAN-Hardware, die autonom lauffähig ist.[11]

2.3.1 Vector XL Library

Die Vector XL Library wird von Vector Informatik als einheitliche Software-Schnittstelle zum Ansprechen der von dieser Firma vertriebenen CAN-, Flexray-, LIN- und MOST-Hardware zur Verfügung gestellt. Sie besteht aus einer dynamischen Bibliothek (DLL) mit zugehöriger Header-Datei, die intern die hardwarespezifischen Treiber der konkret verwendeten Hardware anspricht [Vec].

2.3.2 Port-Konzept

Die zentralen Abstraktionen, die die Vector XL Library zur Verfügung stellt, sind Ports (entspricht einer logischen Verbindung zu einem physikalisch vorhandenen oder virtuellen Kanal, vom Konzept her ähnlich einem SocketCAN-Socket) und abstrakte Ereignis-Objekte, die die unterstützten busspezifischen und generischen Ereignisse (Sende-, Empfangs- und Zeitgeber-Ereignisse) abstrahieren [Vec]. Diese Ereignisse können mit den unter Windows verwendeten Methoden zur Ereignis-Behandlung verwendet werden, wie im nächsten Abschnitt beschrieben. Wie unter SocketCAN, so kann auch mit der Vector XL Library eine Anwendung mehrere Ports für denselben physikalischen Kanal öffnen[12] und die Bibliothek übernimmt die nötige Replikation der Ereignisse und Synchronisation des Zugriffs, und entlastet so den Anwendungsprogrammierer.

[11] Mit dem CAN Case XL steht mittlerweile auch eine autonom lauffähige Hardware zur Verfügung, ähnlich den etablierten Produkten von Samtec oder Berger Elektronik.

[12] Es können auch mehrere Anwendungen einen Port für denselben physikalischen Kanal erzeugen.

Derjenige Port für einen bestimmten physikalischen Kanal, der zuerst geöffnet wird, hat erweiterte Berechtigungen und nur über diesen können Bus-Parameter gesetzt werden [Vec].

2.3.3 Ereignis-Behandlung unter Windows

Windows stellt ein generisches Modell zur Ereignis-Verarbeitung zur Verfügung, das eine einheitliche Behandlung von vom Betriebssystem bereitgestellten Ereignissen und anwendungsspezifischen Ereignissen ermöglicht. Die zentralen Elemente sind die Methoden `WaitForSingleObject` bzw. `WaitForMultipleObjects`, die von der Semantik her dem `select`-Systemaufruf von POSIX-Systemen entsprechen und es ermöglichen, den Kontrollfluss des aktuellen Threads zu suspendieren, um ressourcenschonend auf eines oder eines von mehreren Ereignissen zu warten. Die Ereignisse sind gebunden an einen opaken Datentyp (`HANDLE`), auf dem die Operationen `SetEvent` und `ResetEvent` möglich sind. Wird auf ein solches Objekt die Operation `SetEvent` ausgeführt, so wird ein Thread, der mittels einer der `WaitFor...`-Methoden auf Ereignisse für dieser Objekt wartet, benachrichtigt [13] und nimmt seinen unterbrochenen Kontrollfluss wieder auf. Dadurch ist es für Entwickler, die mit dem low-level Windows-API vertraut sind, einfach, die von der XL Library generierten Ereignisse zu verwerten.

2.3.4 Zeitgeber

Die Vector XL Library stellt, ebenfalls nutzbar über den generischen Ereignis-Behandlungsmechanismus, einen präzisen Zeitgeber zur Verfügung, der durch die verwendete Hardware gesteuert wird, so dass Anwendungen nicht auf die entsprechenden Funktionen des Windows-API angewiesen sind und keinen dedizierten Timer-Thread erzeugen müssen, sondern einen Zeitgeber mit deterministischer, hoher Präzision nutzen können [Vec].

[13] Interessanterweise hat Linux erst verhältnismäßig spät ein API für generische Events eingeführt (unter dem Begriff `eventfd`), das eine ähnliche Semantik bietet wie der beschriebene Mechanismus von Windows. Zuvor mussten benutzerdefinierte Ereignisse mittels generischer Schnittstellen für die Interprozess-Kommunikation wie Pipes oder Message Queues nachgebildet werden (dies ist auch heute noch der bevorzugte Weg, da das `eventfd`-API auf Linux beschränkt ist, Pipes und Message Queues dagegen auf allen POSIX-Systemen vorausgesetzt werden können) [Ker10].

2.3.5 Anbindung von Anwendungen

Von Vector wird die XL Library nur als C-API und als .NET-API (die intern das C-API verwendet, aber das Arbeiten auf einer höheren Abstraktions-Ebene ermöglicht) angeboten; eine Unterstützung von Java müsste mittels JNI realisiert werden, wobei eine vollständige Unterstützung auf Grund des Umfangs sehr aufwändig ist und auf Grund der Verwendung von Out-Parametern auch nicht vollständig äquivalent umgesetzt werden kann. Aus diesem Grund ist der einfachste Weg, die Vector XL Library als zusätzliche Zielplattform zu nutzen, die Verwendung des C-API.

2.3.6 Übersetzung mit `mingw`

Mit dem auf GCC basierenden `mingw`-Compiler steht ein frei redistributierbarer Compiler für Windows zur Verfügung, mit dem sich eine vollständige Werkzeugkette für die Übersetzung der Restbussimulation bei Verwendung des C-API einbinden lässt, ohne auf die Auslieferung eines kommerziellen, mit zusätzlichen Kosten für den Endnutzer verbundenen Compilers angewiesen zu sein.

2.4 Diskussion der Nachteile von SocketCAN und XL Library

Ein Nachteil von SocketCAN liegt darin, dass das zu Grunde liegende Koordinationsmodell, insbesondere das Multiplexen empfangener Botschaften durch den Kernel, nicht mit der regulären 1:1-Beziehung zwischen dem Ereignis „Botschaftsempfang“ und der Reaktion auf die Botschaft in der Anwendung übereinstimmt, wie sie von fast allen anderen APIs zum Ansprechen von CAN-Hardware vorgegeben wird, insbesondere bei mikrocontrollernahen APIs. Diese bieten als minimales API eine synchrone `Can_Send`-Methode sowie einen Benachrichtigung-Mechanismus (mittels Interrupt oder Callback-Funktion) über Empfangs-Ereignisse und eine `Can_Receive`-Funktion zum Auslesen der Botschaft durch die Anwendung nach einer vorangegangenen Benachrichtigung oder durch Polling an; die Möglichkeit, von mehreren Anwendungen aus (oder von mehreren Stellen aus einer Anwendung heraus) auf dasselbe Empfangsereignis zu reagieren existiert bei diesen APIs nicht, sondern muss explizit nachgebildet werden (z.B. durch anwendungsinterne Botschafts-Warteschlangen, wobei die Speicherverwaltung durch die Anwendung erfolgt).

Wird also bei der Anwendungs-Architektur zu sehr auf die von SocketCAN bereitgestellten Mechanismen gesetzt, so beeinträchtigt dies die Portierbarkeit der Anwendungen auf Systeme ohne SocketCAN. Es existieren jedoch APIs, die einen ähnlichen Mechanismus zum Multiplexen der CAN-Kanäle anbieten, z.B. die bereits erwähnte XL Driver Library der Firma Vector, die es ermöglicht, mehrere Anwendungen über so genannte Ports (nicht zu verwechseln mit TCP/IP-Ports) an die eigentlichen CAN-Treiber anzubinden, wobei hier ähnlich wie unter SocketCAN jeder Port eine eigene Empfangsqueue verwaltet. Hier ließe sich vermutlich eine gemeinsame Abstraktionsschicht definieren, die die Schnittmenge der Funktionalität von SocketCAN und XL Driver Library kapselt.

Die speziellen Aspekte der BCM-Sockets, insbesondere das autonome zyklische Senden, lassen sich ebenfalls nicht ohne erheblichen Aufwand auf die oben dargestellten einfacheren APIs portieren, so dass es hier wichtig ist, die Verwendung von BCM-Funktionalität entsprechend zu kapseln. Wird ein AUTOSAR-COM-Stack verwendet, so stehen entsprechende auftragsbasierte Schnittstellen allerdings zur Verfügung [AUTa].

Weiter ist bei den BCM-Sockets kritisch, dass bei der Verwendung des autonomen Sendens keine Möglichkeit besteht, Callbacks vor jedem Senden aufzurufen; für Botschaften, bei denen diese Funktionalität erforderlich ist, muss das zyklische Senden aus der Anwendung heraus erfolgen.

Ein weiteres Problem, dass sich bei zu starker Konzentration auf das durch SocketCAN bereitgestellte Abstraktions-Modell ergibt, ist, dass keine definierten, aus dem User-Space aufrufbaren Schnittstellen zur Konfiguration der Controllers existieren; stattdessen muss das Standard-Tool `ip` verwendet werden, um die CAN-Kanäle der Controller zu konfigurieren [Har]. Die dort vorgenommene Konfiguration beeinflusst das Verhalten aller an diese Kanäle gebundenen CAN-Sockets. Für bestimmte Wechsel der Betriebsmodi des CAN-Controllers (z.B. von der aktiven Teilnahme mit Bestätigung aller empfangenen Nachrichten zum passiven Modus, in dem Nachrichten empfangen, aber nicht bestätigt werden) muss der CAN-Kanal deaktiviert werden und mit veränderten Parametern neu aktiviert werden, was alle daran gebundenen CAN-Sockets ungültig macht; Anwendungen, die dieses Verhalten benötigen, müssen diese Bedingungen abfangen, was die Komplexität der Ereignisbehandlungs-Routinen erhöht.

Die Nachteile der Verwendung der XL Library liegen ähnlich. Auch hier steht das mächtige Port-Konzept eventuellen Portierungen auf eingebettete Plattformen entgegen. Zusätzlich wird die Lösung an einen bestimmten Hardware-Hersteller gebunden und ist nur unter Windows lauffähig; es gibt zwar andere Hersteller, die die Aufruf-Konventionen der XL Library emulieren oder nachbil-

den, aber dies ist eher die Ausnahme, so dass die XL Library in keiner Weise als Standard angesehen werden kann (der hohe Verbreitungsgrad resultiert nur aus der sehr hohen Markt-Präsenz von Vector).

2.5 Alternativen zur beschriebenen Architektur

Zur vorgestellten Architektur gibt es fundamental verschiedene Ansätze, die im Folgenden kurz beschrieben werden.

2.5.1 Echtzeit-Linux

Der für das verwendete Ziel-System verwendete Kernel ist kein Echtzeit-Kernel. Ein Echtzeit-Kernel ist kompilierbar und verwendbar für die CANyon-Hardware, unterstützte zum Zeitpunkt der Arbeit aber wichtige Elemente wie den NAND-Flash-Speicher nicht, so dass dieser aus pragmatischen Gründen nicht produktiv verwendet werden kann. Die prinzipiellen Hintergründe der Echtzeit-Unterstützung unter Linux sind in [Hal11] umrissen.

2.5.2 Kernelmodul für spezifische Restbussimulation

Ein vollkommen anderer Ansatz wäre die Generierung der Restbussimulation und der zugehörigen benutzerdefinierten Erweiterungen als Kernel-Modul. Die stark modularisierte Architektur des Linux-Kernels macht dies möglich, sofern für den verwendeten Kernel das dynamische Laden von Modulen [14] aktiviert ist. Diese Architektur-Variante ist in Abbildung 2.11 dargestellt. Bei Verwendung eines Kernels, der keine dynamischen Module unterstützt, könnte ebenfalls die Restbussimulation als Kernel-Modul generiert werden, dieses müsste dann jedoch statisch gegen den Kernel gelinkt werden, was zur Folge hätte, dass immer der komplette Kernel neu auf das Zielgerät aufgespielt werden müsste, was aus verschiedenen Gründen (Dauer des Vorgangs und die Gefahr, das System in einen unbenutzbaren Zustand zu bringen) nicht akzeptabel ist. Die Vorteile dieser Lösung sind:

- Zero-Copy möglich, d.h. bestmögliche Vermeidung von Puffer-Kopien von empfangenen und gesendeten Datenpaketen. Da Restbussimulation und Kernel im gleichen Adressraum ablaufen, können Referenzen auf Datenpakete (mittels Zeigern) zwischen den Schichten weitergeleitet werden

[14] Dynamische Kernel-Module sind keine dynamischen Bibliotheken (`libXX.so`) im Sinne des unter Linux üblichen `ld`-Linkers, wie man vermuten könnte.

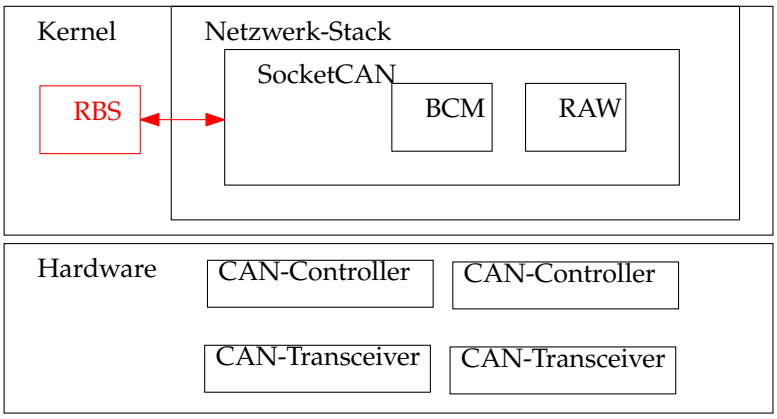

Abbildung 2.11: Architektur-Alternative: Restbussimulation als Kernel-Modul. Dies vermeidet jegliche Userspace-Kernelspace-Puffer-Kopien durch die Kommunikationstätigkeiten der RBS. [Ber]

anstatt (beim Übergang zwischen Kernel und User-Space) Kopien erzeugen zu müssen.

- Bestmögliches Timing, da Kernel-Timer-Callbacks genutzt werden können und keine Latenz durch das Aktivieren eines auf ein Timer-Ereignis wartenden User-Space-Threads durch den Scheduler auftritt.
- Die Nutzung hardwarespezifischer Funktionen, z.B. der CAN-Mailboxen auf dem Controller unter Umgehung des SocketCAN-API, wird ermöglicht. Somit kann die bestmögliche Ausnutzung der Fähigkeiten der Hardware gewährleistet werden.

Zu den Nachteilen zählen:

- Erschwertes Debugging. Da die Restbussimulation im Kernel abläuft, kann kein einfaches Remote-Debugging (z.B. mit `gdb` und `gdbserver`) durchgeführt werden, sondern es müssen spezielle Kernel-Debugger eingesetzt werden. Zusätzlich müssen in Abhängigkeit davon, in welcher Schicht genau das Problem vermutet wird, bestimmte Kernel-Optionen aktiviert werden, was unter Umständen das Neuübersetzen des Kernels erfordert. [15]

[15] Die Aktivierung aller Debug- und Trace-Optionen des Kernels führt, sofern der Kernel damit überhaupt übersetzbar ist, mit Sicherheit zu interessantem Laufzeit-Verhalten, trägt aber in aller Regel nicht zur Identifikation des Problems bei.

- Dem Benutzer, der die benutzerdefinerten Teile der Restbussimulation implementiert, wird extreme Programmierdisziplin abverlangt, um blockierendes Verhalten zu vermeiden. Innerhalb eines Kernel-Moduls hat blockierendes Verhalten unter Umständen sehr obskure, schwierig zu ermittelnde Ursachen einerseits und Symptome andererseits. Des Weiteren steht innerhalb des Kernels die C-Standard-Bibliothek (`libc` unter Linux) nicht zur Verfügung, so dass zwangsläufig kernelinterne APIs genutzt werden müssen, was die Portabilität des benutzerdefinierten Teils der Restbussimulation beeinträchtigt.

- Lizenzproblematik: der Kernel steht unter der GPL, daher muss äußerst streng darauf geachtet werden, welche Kernel-Funktionen verwendet werden können und wie gegen den Kernel gelinkt wird, um eine Kontamination eines proprietären RBS-Moduls zu vermeiden. [16]

Die beiden zuerst aufgeführten Nachteile sind sehr schwerwiegend, da die Restbussimulation (als Artefakt einer Rapid-Prototyping-Werkzeugkette) einfach zu entwickeln sein muss; spezielle Erfahrung im Debuggen von Kernel-Modulen oder spezifische Kenntnisse über kernelinterne Kommunikationsmechanismen und APIs und deren Präemptionsverhalten gehören typischerweise nicht zum Repertoire eines Prüfstandsingenieurs.

2.5.3 Eingebetteter Interpreter

Ein anderer Ansatz, die Erweiterung von Restbussimulationen durch den Benutzer zu ermöglichen, besteht darin, ein statisches Restbussimulations-Framework zur Verfügung zu stellen, das einen Skript-Interpreter einbettet. Sämtliche benutzerspezifischen Aspekte (sowohl die Kommunikations-Matrix als auch benutzerdefinierte Logik und Ereignisbehandlung) kann dann als Quelltext in der Sprache des verwendeten Interpreters auf des Zielsystem geladen werden. Mögliche Kandidaten sind Lua und Python, die beide auf der eingesetzten CANyon-Hardware lauffähig sind und sich leicht in eine Host-Anwendung in C einbetten lassen sowie ein leistungsfähiges Foreign-Function-Call-Interface für die Integration beliebiger C-Funktionen bieten. Für Lua gibt es Ansätze für einen Just-in-Time-Compiler für die PowerPC-Architektur, so dass hier u.U. auf dem Zielsystem selbst der Quellcode zur Laufzeit in Maschinencode kompiliert werden kann anstatt die

[16] Bei statisch gegen den Kernel gelinkten Modulen trifft dies immer zu. Da die RBS-Anwendung jedoch nicht ausgeliefert wird, sondern vom Kunden generiert wird und in der Regel nicht vertrieben, sondern intern genutzt wird, befindet man sich rechtlich in einer (hellen) Grauzone.

Performance-Einbußen durch die Interpretation von Bytecode in Kauf nehmen zu müssen. Die Vorteile einer solchen Lösung sind:

Kein Cross-Compiler Es genügt ein einfacher Upload der Quelldateien auf das Zielsystem; der Benutzer benötigt keinen Cross-Compiler, der den Quellcode auf dem Host-System übersetzt.

Einfachere Sprache Die erwähnten Skriptsprachen erfordern u.U. weniger Quellcode im Vergleich zum benötigten C-Code für äquivalente Aufgaben. Das dynamische Redefinieren von Funktionen ist einfacher möglich als es in C z.B. durch die Verwendung von Funktionszeigern ist.

Der Interpreter-Ansatz weist auch einige Nachteile auf:

Performance Der zu erzielende hohe Datendurchsatz ist mit interpretierten Sprachen (selbst wenn es sich um interpretierten Bytecode wie bei Python und Lua handelt) schwieriger zu erzielen.

Portierbarkeit Der Interpreter stellt signifikante Anforderungen an die Laufzeitumgebung. Python ist ohne Änderungen nicht ohne ein Betriebssystem auf der Zielplattform ausführbar. Für Lua wird zumindest eine C-Laufzeitumgebung vorausgesetzt, die unter Umständen mächtiger ist als das, was kleinere eingebettete Systeme bieten.

Sprache Die Einarbeitung in eine andere Programmiersprache ist notwendig, was in manchen Umgebungen nicht akzeptabel ist.

Tooling Das Tooling für den verwendeten Interpreter bzw. dessen Sprachfamilie ist u.U. nicht ausreichend für den kommerziellen Einsatz; besonders schwer wiegt hier in der Regel das Fehlen eines Remote-Debuggers.

Zu den kommerziellen Anbietern, die diesen Ansatz wählen (nicht dediziert für den Bereich Restbussimulationen, sondern im Bereich HIL-Simulation) [17], gehören unter anderem DSpace, die einen echtzeitfähigen Python-Interpreter für ihre Produkte DS1005 PPC und DS1006 bereitstellen.

2.6 Anwendbarkeit von AUTOSAR

Die Anforderungen an ein System zur Restbussimulation unterscheiden sich teilweise deutlich von Anforderungen an Architekturen für einzelne ECUs zur

[17] Hardware-in-the-Loop

Verwendung in der Serienproduktion eines Gesamtfahrzeugs, und dies nicht nur im Hinblick auf die offensichtlich differierenden nicht-funktionalen Anforderungen, sondern auch in Bezug auf konkrete funktionale Anforderungen. Die Unterschiede lassen sich am plausibelsten erklären, indem man die Restbussimulation als Produkt eines Rapid-Prototyping-Prozesses auffasst, der, wie beim Rapid Prototyping üblich, hohe Produktivität für bestimmte Klassen von Problemen ermöglicht, mit dem Kompromiss, dass Teilaspekte der erzielten Lösungen weniger optimiert sind als in einer vollständig durchkonfigurierten anwendungsspezifischen Einzellösung.

Die AUTOSAR-Architektur, geschaffen als herstellerübergreifendes Instrument zur System-Integration zwischen Gesamtfahrzeugherstellern und deren Zulieferern, hat den Schwerpunkt, die Integration von Modulen mit nahezu vollständig statisch definierten Schnittstellen zu steuern und zu vereinfachen [KF09], wohingegen Restbussimulationen unter anderem eingesetzt werden, um dynamisch Eigenschaften eines Gesamtsystems auf einfache Weise variieren zu können, um das Verhalten auf diese Variationen analysieren zu können.

Bestimmte Teile von AUTOSAR, insbesondere die dort vorgegebenen Schichten und deren Abstraktionsniveau, sind im Kontext der Architektur für Restbussimulationen durchaus anwendbar, unter Umständen mit Einschränkungen, wie im Folgenden beschrieben, während andere Aspekte sich nicht sinnvoll auf Architekturen für Restbussimulationen übertragen lassen.

Aus diesem Grund erschien es sinnvoll, in den folgenden Abschnitten zunächst eine kurze Einführung in das AUTOSAR-Konzept zu geben, bevor abschließend in Abschnitt 2.6.8 die Konflikte zwischen dem Abstraktionsgrad von AUTOSAR und dem Abstraktionsgrad einer Restbussimulation diskutiert werden.

2.6.1 Duale Funktion

AUTOSAR hat eine duale Funktion bei der System-Integration bei Gesamtfahrzeug-Herstellern und deren Zulieferern: zum einen definiert es eine Methodologie, ein Vokabular und eine Notation für den Entwurf und die Beschreibung von Software-Komponenten und deren Interaktionen. Andererseits legt es ein (konservatives) Schichten-Modell für die Interaktion zwischen Software-Komponenten und so genannter Basis-Software (BSW; siehe Abschnitt 2.6.5) fest. Diese beiden Aufgaben erfüllt es sowohl im Kontext des Gesamtfahrzeugs als auch im Kontext des einzelnen Steuergerätes, d.h. es wird eine Überführung von der Architektur für das Gesamtsystem in die Architektur des Steuergerätes ermöglicht und für die dabei entstehenden bzw. auftretenden Integrationsaufga-

ben definierte Lösungsmuster definiert. Die steuergerätebezogene Sicht wird im Allgemeinen als ECU-zentrische Sicht bezeichnet; separierte Teile einer Gesamtsystem-Spezifikation, die die für eine einzelne ECU relevanten Informationen enthalten, werden analog dazu als ECU-Extrakte bezeichnet [KF09].

2.6.2 Ablauf

Die oben genannte Dualität schlägt sich auch in der Rollenverteilung nieder, wobei es sinnvoll ist, zwischen Entwickler und Integrator zu unterscheiden; der Ablauf entspricht dabei folgendem Muster: der Entwickler erhält die Schnittstellen-Spezifikation der zu entwickelnden Komponente in Form eines Satzes von AUTOSAR-Dokumenten (zusammen mit anderen, nicht nach AUTOSAR formalisierten funktionalen und nicht-funktionalen Anforderungen), und setzt die Komponente um (interner Entwurf, Implementierung und Test). Der Integrator erhält die Komponente (entweder als Quellcode oder als kompilierten Objektcode für eine bestimmte Hardware-Architektur) und kann diese durch die im Zusammenhang mit der formalen Schnittstellen-Beschreibung definierten Konfigurations-Schnittstelle (siehe Abschnitt 2.6.7) in ein Gesamtsystem integrieren.

2.6.3 Komponenten-Modell

Für die Modellierung von Software-Komponenten und deren Schnittstellen sowie für die Modellierung der Verbindungen zwischen konkreten Instanzen dieser Komponenten stellt AUTOSAR eine an der UML orientierte Notation bereit, die in serialisierter Form als so genannte AUTOSAR-Software-Component vorliegt und in ein AUTOSAR-System eingebunden werden kann. Die Details dieser Notation werden in [AUTc] beschrieben.

Diese Notation kann, wie oben erwähnt, zur Beschreibung eigener Software-Komponenten verwendet werden; jedoch ist auch die Basis-Software, d.h. die von AUTOSAR bereits innerhalb des Standards vordefinierten Module, in dieser Notation beschrieben, so dass der Zugriff von eigenen Komponenten und Anwendungs-Software auf die Basis-Software von AUTOSAR einfach modelliert werden kann und sichergestellt werden kann, dass nur die von der Basis-Software bereitgestellten Schnittstellen verwendet werden.

Hinsichtlich der Werkzeug-Unterstützung von AUTOSAR hat die einheitliche Beschreibung von Komponenten der Basis-Software und benutzerdefinierten Software-Komponenten den weiteren Vorteil, dass Standard-Komponenten (BSW-Module) und benutzerdefinierte Komponenten auf einheitliche Weise mit

Werkzeugunterstützung konfiguriert werden können; gesonderte Konfigurations-Werkzeuge für die benutzerdefinierten Software-Komponenten entfallen.[18] Dies hat auch Auswirkungen auf die Code-Generatoren, die die Anpassung der Software-Komponenten realisiert, da auch hier für Basis-Software und eigene Module ein einheitliches Konzept verwendet werden kann.

2.6.4 Virtual Functional Bus

AUTOSAR führt mit dem so genannten *Virtual Functional Bus (VFB)* eine Abstraktions-Schicht für die Anbindung von Signalen an die Software-Komponenten ein, die von der konkreten Übertragungsform dieser Signale abstrahiert (d.h. auf hohen Modellierungs-Ebenen ist nicht beschrieben, mittels welchem Medium oder welchem Bus-Typ Signale zwischen Komponenten propagiert werden). Diese Abstraktion kann in nachfolgenden, feineren Modellierungs-Schritten aufgehoben werden, indem sogenannte Signalpfad-Beschränkungen eingeführt werden, mittels derer explizit angegeben werden kann, über welche physikalischen Kanäle (CAN, LIN, Flexray, Ethernet, ECU-intern) das Weiterleiten von Signalen erfolgen muss, erfolgen darf oder nicht erfolgen darf. Der Haupt-Grund für die Notwendigkeit für diese Beschränkungen liegt darin, dass unter Umständen manche bestehende Geräte nicht in der AUTOSAR-Systembeschreibung erfasst sind; dies trifft z.B. für spezielle Steuergeräte zu, die nicht Teil der Serie sind, wie etwa Steuergeräte, die Teil der Messausrüstung sind und nur für Fahrzeuge der Testflotte relevant sind.

Konkret ist der Virtual Functional Bus eine signalorientierte Schnittstelle, die das Setzen und Lesen von Signal-Werten in einer Weise ermöglicht, die die Komponenten, die diese Schnittstelle verwenden, vom konkret verwendeten Signalübertragungs-Mechanismus unabhängig macht. In nachgelagerten Schritten erlaubt der AUTOSAR-Prozess, dass so z.B. Signale, die zwischen Komponenten einer einzelnen ECU ausgetauscht werden, rein über den Zugriff auf gemeinsame Variablen zwischen den betroffenen Komponenten realisiert wird (bei der Generierung der RTE für die entsprechende ECU) und keine Signalübertragung über externe Kanäle erfolgt.

[18] Für ein sehr komplexes Modul tritt unter Umständen der Fall ein, dass dieses zwar theoretisch in generischer Weise konfiguriert werden kann, dies jedoch nicht praktikabel ist, so dass zusätzlich eine spezielle Konfigurationslösung nötig ist. Für Module, die lediglich einfache Parametrisierungen erfordern, ist der generische Ansatz jedoch ausreichend.

2.6.5 Basis-Software

Mit dem Begriff Basis-Software (BSW) werden unter AUTOSAR bestimmte Software-Komponenten bezeichnet, deren funktionale Anforderungen und Schnittstellen durch AUTOSAR selbst in Form einer Spezifikation vorgegeben werden. Dies sind in der Regel Abstraktionen für Treiber und ergänzende Abstraktionen für die Kommunikation zwischen Anwendungs-Schicht und diesen Treibern. Ein weiterer großer Teil der Basis-Software besteht aus dem Kommunikations-Stack und dessen unterstützenden und ergänzenden Modulen (PDU-Router, PDU-Multiplexer etc.) [AUTa].

2.6.6 Runtime Environment

AUTOSAR erwartet auf einer konkreten ECU, auf der AUTOSAR-Komponenten oder Basis-Software laufen sollen, dass dort eine Laufzeitumgebung, die sogenannte *Runtime Environment (RTE)* vorhanden ist. Diese wird durch einen ECU-spezifischen Generator generiert, zusammen mit ECU-spezifischen Varianten der Basis-Software. Die RTE entspricht der ECU-zentrischen Sicht auf das System. Hier wird der Signal-Fluss, der über das VFB-Konzept modelliert wurde, konkret umgesetzt, indem der Generator für die RTE entscheidet, ob Signale an Software-Komponenten auf andere ECUs weitergeleitet werden müssen oder nur zur Kommunikation zwischen Software-Komponenten innerhalb dieser ECU verwendet werden. Im letzteren Fall können die bereits im Abschnitt zum VFB erwähnten Optimierungen bezüglich des Signal-Zugriffs durchgeführt werden; andernfalls müssen diese Signale mittels des Kommunikations-Stacks über die der ECU zur Verfügung stehenden physikalischen Kanäle weitergeleitet werden.

2.6.7 Konzept der Konfiguration

Das Prinzip, die Schnittstellen zur Konfiguration einer Komponente in die formale Schnittstellen-Definition mit aufzunehmen sowie die Differenzierung zwischen unterschiedlichen Ebenen und Zeitpunkten für die Konfiguration ist konzeptuell dasjenige Kriterium, das AUTOSAR im Umfeld der Symbiose zwischen Zulieferer und Gesamtfahrzeug-Hersteller besonders attraktiv macht. Beschreibungs-Methoden für Software-Komponenten gibt es in großer Zahl und mit hoher Varianz hinsichtlich der Praxistauglichkeit, aber ein ähnlich detailliertes Konfigurations-Modell als Kernaspekt (mit dem Fokus auf die Compiler- und Linker-Spezifika von C-Modulen im Hinblick auf eingebettete Systeme) findet man nur selten,

wie ein Vergleich von AUTOSAR mit den in [TMD09] vorgestellten Architektur-Modellen leicht ergibt. [19]

2.6.8 Konflikte zwischen AUTOSAR-Abstraktionen und Restbussimulationen

Ein Konfliktpunkt zwischen einer Architektur im Sinne von AUTOSAR und einer Architektur für Restbussimulationen liegt darin, dass eine konkret realisierte AUTOSAR-Implementierung immer an eine ECU-Instanz gebunden ist. Eine Restbussimulation hingegen soll in der Regel mehrere ECUs auf einer gemeinsamen Hardware simulieren. Eine Hardware-Abstraktion im Sinne von AUTOSAR lässt sich hier nicht vollständig simulieren, da bestimmte Funktionen (z.B. Ändern der Betriebs-Modi von CAN-Controllern) das Verhalten bestimmter Hardware-Module beeinflussen müssen, um extern sichtbare Auswirkungen zu haben; soll die Restbussimulation (die mehrere ECUs simuliert) z.B. den Übergang einer ECU von aktiver Teilnahme am CAN-Bus zu passiver Teilnahme simulieren, so beeinflusst dies auch alle anderen simulierten ECUs.

Eine andere Quelle von Unterschieden zwischen Restbussimulationen und dem AUTOSAR-Konzept liegt in der starken Fokussierung von AUTOSAR auf statische Bindung und Zuweisung. Für den schnellen Entwicklungszyklus, der für eine Restbussimulation typisch ist, muss es möglich sein, bestimmte Konfigurationseinstellungen, die von AUTOSAR statisch vorgenommen werden, zur Laufzeit oder zumindest über eine Post-Build-Konfiguration vorzunehmen, da eine vollständige Neugenerierung der Anwendung die Produktivität des Anwenders empfindlich stört.[20]

Ein weiterer Aspekt, der die Entwicklung von Restbussimulationen mittels der AUTOSAR-Methodik erschwert, ist die stark signalorientierte Herangehensweise, die bei der Entwicklung und Einbindung von Software-Komponenten in ein AUTOSAR-System notwendig ist; traditionelle Werkzeuge zur Erstellung von Restbussimulationen sind noch stark an den busspezifischen Botschaften orientiert und betrachten Signale nicht auf Systemebene, sondern als Teile der gesendeten Botschaften. AUTOSAR abstrahiert auf der Schnittstellenebene zwischen Software-Komponenten von den konkreten Botschaften, so dass eine botschaftsorientierte Entwicklung von Funktionalität, wie sie in der tra-

[19] Diese Spezialisierung auf C führt natürlich dazu, dass argumentiert werden kann, AUTOSAR sei gegenüber anderen Architekturbeschreibungs-Sprachen zumindest in Teilen weniger allgemeingültig.

[20] Exemplarisch für die Problematik der Konflikte zwischen dynamischen APIs und statischer Konfiguration sei das SocketAdaptor-Modul genannt, das zu dem Zweck existiert, die dynamische Natur von TCP/IP-Sockets auf die unter AUTOSAR übliche Weise statisch konfigurierbar zu machen.

ditionellen Entwicklung von Restbussimulationen vorherrschend ist, erschwert wird [KF09]. Symptomatisch hierfür ist zum Beispiel, dass eigene Software-Komponenten über so genannte Ports eingebunden werden müssen, wenn sie Daten von der AUTOSAR-RTE benötigen, und diese Ports die Semantik von Signalen widerspiegeln.

Kapitel 3

Modulare Code-Generierung

Code-Generierung ist prinzipiell immer dann notwendig, wenn eine grundsätzlich hinreichend genaue formale Spezifikation [1] auf Grund von Unzulänglichkeiten der verwendeten Programmiersprache oder Beschränkungen der Laufzeitumgebung nicht mit vertretbarem Aufwand direkt in ausführbare Form gebracht werden kann. Aufgabe der Code-Generierung ist es, eine zusätzlich Abstraktionsebene zu schaffen, die statt der Transformation von der Spezifikation zum Zielsystem eine Transformation von der Spezifikation in ein Zwischenformat und von diesem Zwischenformat ins Zielsystem realisiert. Beim klassischen Begriff der Code-Generierung aus dem Compilerbau (der hier nicht betrachtet wird) werden für diese unterschiedlichen Ebenen die Begriffe Frontend (Verstehen der Spezifikation, im Compilerbau sind dies Programmiersprachen), Middleend (ein Neologismus für compilerabhängige Zwischendarstellungen, im Compilerbau sind dies Datenstrukturen für abstrakte, annotierte Syntaxbäume oder linearisierte Darstellungen davon) und Backend (im Compilerbau meist spezifischer Assembler- oder Maschinencode für bestimmte Prozessorarchitekturen, seltener Hochsprachen) verwendet. Bei der Code-Generierung im in dieser Arbeit betrachteten Sinn ist das Frontend zuständig für das Verständnis von Busbeschreibungsdateien und das Backend zuständig für die Generierung von Hochsprachen-Code, der die in der Spezifikation beschriebenen Kommunikationsmuster für eine bestimmte Klasse von Zielsystemen realisiert.

[1] d.h. eine Spezifikation, die eine bestimmte Problemdomäne vollständig beschreibt

3.1 Typische Ansätze

Im Folgenden werden übliche Ansätze für die Code-Generierung auf unterschiedlichen Ebenen kurz vorgestellt, wobei der Fokus auf einer möglichst breiten Sicht liegt.

3.1.1 Analogien zum Compilerbau

Die in der Einleitung zu diesem Kapitel beschriebene Verteilung der Aufgaben auf drei Schichten weist neben der allgemein sinnvollen Trennung der Zuständigkeiten den zusätzlichen Vorteil auf, dass bei Unterstützung mehrerer Eingabe- und Ausgabeformate kein kombinatorisch ansteigender Aufwand betrieben werden muss, sondern (im Idealfall) jedes Frontend mit jedem Backend verwendet werden kann, da die Transformation in eine Zwischendarstellung diese beiden Schichten entkoppelt: für die Unterstützung neuer Eingabeformate kann ein neues Modul eingeführt werden, das die Transformation in die Zwischendarstellung realisiert; die Transformation in das Ausgabeformat verwendet die bestehenden Module. Für die Unterstützung neuer Ausgabeformate muss nicht eine Transformation von jedem Eingabeformat in das neue Ausgabeformat implementiert werden, sondern lediglich eine Transformation von der Zwischendarstellung in das neue Ausgabeformat. Für n Eingabeformate und m Ausgabeformate reduziert sich so die Anzahl der zu implementierenden Transformationen von $n \times m$ zu $n + m$.

Der AUTOSAR-Ansatz unterscheidet sich von diesem Ansatz in wesentlichen Punkten:

- Es wird ein kanonisches Eingabeformat vorgeschrieben (die AUTOSAR-Systembeschreibung); eine Unterstützung von anderen Eingabeformaten muss zwangsläufig außerhalb des AUTOSAR-Kontexts erfolgen (z.B. durch eine externe Transformation in AUTOSAR-konforme Fragmente). Eine gesonderte Zwischendarstellung entfällt damit.

- Es muss eine AUTOSAR-Laufzeitumgebung für die Zielplattform existieren. Große Teile der Laufzeitumgebung werden zwar generiert, dies setzt allerdings die Existenz der entsprechenden AUTOSAR-Basissoftware für die Zielplattform voraus.

AUTOSAR muss diese Einschränkungen vornehmen, da das abgedeckte Anwendungsgebiet viel breiter ist als das in der vorliegenden Arbeit abgedeckte Gebiet der Restbussimulation und weil eine AUTOSAR-System-Spezifikation die

Definition von Aspekten erlaubt, die spezifische Interna der Laufzeitumgebung betreffen (es wäre beispielsweise sinnlos, in der Spezifikation ein bestimmtes Scheduling-Verhalten definieren zu können, ohne dass bekannt ist, ob und wie dieses von der Laufzeitumgebung unterstützt wird und wie die Abbildung auf das Laufzeitsystem erfolgen muss). Für die Generierung einer Restbussimulation ist lediglich die Garantie notwendig, dass für eine Zielplattform das genau definierte (und eingeschränkte) API zur Verfügung steht und die versprochene Semantik hat; damit sind die Anforderungen an die Zielplattform wesentlich übersichtlicher, so dass insgesamt die Komplexität der Informationen, die vom Frontend geliefert und durch das Middleend transformiert werden müssen, viel geringer ist.

3.1.2 Sprachinterne Code-Generierung

Als transzendenter Idealfall des Software-Engineering kann die Transformation einer Spezifikation in ein ausführbares System innerhalb einer Sprache gelten. Für eine solche Sprache muss gelten, dass das Abstraktionsniveau nicht vorgegeben ist, sondern vom Benutzer aufgabenspezifisch angepasst werden kann. Dies wird meist so realisiert, dass für bestimmte syntaktische Elemente die sonst übliche Interpretation durch Parser oder Compiler aufgehoben wird und durch vom Benutzer definierte Logik ersetzt wird. Dadurch lässt sich realisieren, was in den letzten Jahren unter dem Begriff eingebettete domänenspezifische Sprachen (DSL) zusammengefasst wird, nämlich die Möglichkeit, beliebige strukturierte Informationen innerhalb der Sprache zu definieren, diese ebenfalls mit Mitteln der Sprache bzw. deren Laufzeitumgebung zu transformieren und die transformierte Darstellung dynamisch zu interpretieren. [2] Charakteristisch ist hierbei, dass die erweiterten Elemente auf symbolischer Ebene und nicht auf lexikalischer Ebene interpretiert werden, was einerseits die Möglichkeiten einschränkt und andererseits die Interpretation erleichtert. [3]

Zu den bekanntesten Sprach-Familien, die derartiges ermöglichen, gehören die Lisp- und Scheme-Familien und deren Derivate sowie bestimmte stackbasierte Sprachen wie Factor.

[2] Wobei der Begriff „Interpretation“ hier eine eventuelle Just-In-Time-Kompilierung nicht ausschließt.

[3] In Common Lisp existiert allerdings die Möglichkeit, Erweiterungen auf lexikalischer Ebene durch so genannte Reader-Erweiterungen zuzulassen, wobei der Reader hier der Common Lisp spezifische Mechanismus für das lexikalische Scannen ist, dessen Funktionsumfang über die sonst während der lexikalischen Analysephase durchgeführten Aufgaben hinausgeht. So können etwa benutzerdefinierte Callback-Funktionen aufgerufen werden, die beim Auftreten bestimmter lexikalischer Elemente in einen benutzerdefinierten Lexer-Modus umschalten.

Eine größere Akzeptanz dieser Sprachfamilien in der Industrie blieb bisher aus; über Konzepte wie aspektorientierte Programmierung durch Bytecode-Weaving zur Laufzeit diffundieren diese Mechanismen jedoch (zum Teil in stark eingeschränkter Form) in die in der Industrie etablierten Sprachfamilien der JVM- und .NET-Plattformen oder sind in Form des so genannten Monkey-Patchings in dynamisch typisierten Sprachen wie Python und Ruby wiederzuerkennen. [4] In Java muss zur dynamischen Generierung von Bytecode auf externe Bibliotheken wie ObjectWeb ASM oder `cglib` zurückgegriffen werden, während unter .NET mit `System.Reflection.Emit` ein Modul zur dynamischen Bytecode-Generierung bereits Teil der Standard-Bibliothek ist. [5] Auch in C gibt es isolierte Beispiele für (Maschinen-) Code-Generierung zur Laufzeit, z.B. im Kapitel von Petzold in [OW07].

Es gibt viele technische Gründe, die einer Implementierung von derartigen Transformations-Aufgaben innerhalb einer geeigneten Sprache entgegenstehen; die häufigsten sind:

- Für die Spezifikation des Problems steht bereits ein extern vorgegebenes Beschreibungsformat fest (Standard oder Quasi-Standard), so dass eine Spezifikation des Problems als eingebettete DSL in dieser Hinsicht keine Vorteile bringt. Im Kontext dieser Arbeit äußert sich dies in der Existenz bestehender Beschreibungsformate für die Kommunikations-Matrix.

- Für ein Zielsystem steht keine bzw. keine ausreichend performante Laufzeitumgebung für die Sprache zur Verfügung, oder der Ressourcenbedarf eines derartigen Laufzeitsystems übersteigt die zulässigen extern definierte Beschränkungen des Zielsystems. Im Kontext dieser Arbeit kommt dieser Punkt zum Tragen, da eine Portierung auf kleiner dimensionierte Hardware-Plattformen möglich sein muss.

- Es muss eine Verknüpfung der generierten Funktionalität mit bestehenden System-Komponenten erfolgen. Im Kontext dieser Arbeit ist dies die Ver-

[4] Im Falle der aspektorientierten Programmierung war der Verantwortliche für die bekannteste Implementierung (AspectJ) gleichzeitig der Mitgestalter des Meta-Object-Protokolls von Common Lisp.

[5] Seit Version 1.6 des Sun (bzw. jetzt Oracle) Java Development Kit gibt es das Compiler-API (im Paket `javax.tools`), das das dynamische Kompilieren von Java-Quelldateien in Bytecode und das Laden der generierten Klassen realisiert, indem es den bestehenden `javac`-Compiler innerhalb einer laufenden JVM über interne Mechanismen ansprechbar macht. Allerdings ist dies eher als Notlösung zu verstehen, da dies sich prinzipiell nicht von einem Aufruf des `javac`-Compilers als externen Prozess unterscheidet.

knüpfung mit bestehenden Betriebssystem-Schnittstellen oder Hardware-Treibern.

- Die generierte Grund-Funktionalität muss in bestimmter Weise erweiterbar sein durch benutzerdefinierte Logik. Im Kontext dieser Arbeit ist dies die Einschränkung, dass eine generierte Restbussimulation durch ein C-API vom Benutzer erweitert werden können muss.

Keines dieser Argumente ist ein Ausschlußkriterium. Allerdings wird die Durchgängigkeit, die eine sprachinterne Lösung auszeichnet, unterbrochen.

Häufig angeführte Argumente, die gegen die Verwendung solcher Lösungen aus Gründen der Verständlichkeit für den Entwickler sprechen, basieren meist auf dem falschen Vergleich zwischen einer Mainstream-Sprache (für die berechtigterweise keine Einarbeitung vorausgesetzt wird) und der Lösungs-Sprache (für die der Einarbeitungsaufwand berechtigt als hoch angesetzt wird); es müsste jedoch statt der alleinigen Betrachtung der Mainstream-Sprache eine Betrachtung der Mainstream-Sprache und allen an der Transformation beteiligten Technologien erfolgen und die Einarbeitung in diese Technologien mitberücksichtigt werden. Für eine komplexere Transformationskette kann dieser Aufwand erheblich sein. Ebenso muss für die Betrachtung der Wartbarkeit die Komplexität betrachtet werden, die eine aus mehreren Technologien mit unterschiedlichen Entwicklungszyklen bestehende Lösung mit sich bringt. Hinzu kommt die häufig trivialisierte Tatsache, dass in komplexeren Transformationsketten effektiv mehrere Programmiersprachen zum Einsatz kommen. Als Beispiel sei XSLT genannt, das eine Turing-vollständige, eigene Sprache ist. [6] Ein anderes Symptom ist die Verwendung von komplexen Mapping-Definitionen in technologiespezifischen XML-Formaten, deren Umfang bei reinen Transformationsprojekten häufig den Umfang der eigentlichen Implementierung in der Host-Sprache übersteigt, so dass die Aufgaben der Host-Sprache auf einfachen Glue-Code reduziert werden. Argumente, die sich für die Host-Sprache auf Grund von Aspekten wie statischer Typsicherheit aussprechen, verlieren empfindlich an Glaubwürdigkeit, wenn der Großteil der Domänenlogik in untypisierten Textdateien definiert ist. [7]

[6] Zur Betrachtung von XSLT als funktionale Programmiersprache siehe `http://fxsl.sourceforge.net/articles/FuncProg/Functional%20Programming.html`

[7] Zwar sind XML-Formate in der Regel durch DTD- oder XSD-Beschreibungen definiert, allerdings sind diese häufig nur unvollständig oder ad-hoc mit dem Typsystem der Host-Sprache verbunden. Technologien wie EMF [Ecld] oder JAXB [jax] beheben diese Diskrepanzen zum Teil, indem Klassen generiert werden, die die durch das Schema definierten Beziehungen ins Typsystem der Host-Sprache abbilden und so eine ganzheitliche Typisierung schaffen.

3.1.3 Sprachübergreifende Code-Generierung

Üblicher als die im vorigen Abschnitt beschriebene sprachinterne Code-Generierung ist die sprachübergreifende Code-Generierung, d.h. das Erzeugen von Code-Artefakten einer Zielsprache mittels eines Generators, der in einer bestimmten Host-Sprache implementiert ist.

Hierbei gibt es prinzipiell drei Ausprägungen, wie die Generierung erfolgen kann:

Direkt Die Ein- und Ausgabemechanismen der Host-Sprache können prinzipiell verwendet werden, um beliebige Artefakte zu erzeugen. Lösungen dieser Art sind aus einer Vielzahl von Gründen zum strukturierten Generieren größerer Artefakte nicht praktikabel: potentiell schlechte Wartbarkeit durch fast zwangsläufig erfolgende Vermengung von Datenaufbereitung und Datenausgabe, manuelle Iteration über Eingangsdaten notwendig, hohe Fehleranfälligkeit durch fehlende frühe syntaktische Prüfung der Ausgabe, Notwendigkeit der Übersetzung vor dem Ausführen, Erweiterungen durch den Endnutzer nicht einfach möglich.

Template-basiert Der in der industriellen Praxis am häufigsten anzutreffende Ansatz ist eine duale Lösung, die aus einem Treiber in einer Host-Sprache und der Spezifikation der Ausgabe mittels einer Template-Sprache besteht; eine Diskussion der Vorteile dieser Lösung erfolgt im Anschluss.

AST-basiert Wenn die zu generierenden Artefakte einer bestimmten bekannten Syntax entsprechen (was in vielen Fällen vorausgesetzt werden kann), so kann das zu generierende Artefakt als abstrakter Syntaxbaum (*Abstract Syntax Tree*, AST) erzeugt und manipuliert werden; zur tatsächlichen Manifestation des Artefaktes als konkrete Datei kann der abstrakte Syntaxbaum dann serialisiert werden. Voraussetzung ist, dass ein Mechanismus zur Verfügung steht, um die Syntax als abstrakten Syntaxbaum zu repräsentieren, z.B. im Falle von C als Klassen-Bibliothek, die die AST-Knoten als programmatisch manipulierbaren Objektgraph zugänglich machen. Ein solcher Objektgraph kann dann serialisiert werden, um das zu generierende Artefakt zu erzeugen. Diese Lösung bietet den Vorteil, dass die syntaktische Korrektheit des generierten Artefakts garantiert werden kann. Insbesondere bei mehrstufigen Manipulationen am Ausgabeformat erhöht dieser Ansatz so die Zuverlässigkeit. Ein weiterer Vorteil ist die leichtere Testbarkeit: wenn das zu generierende Artefakt als Objektgraph programmatisch zugänglich ist, können beliebige Aspekte leicht program-

matisch überprüft werden; bei anderen Lösungen muss zu Testzwecken das generierte Artefakt textuell untersucht werden, was aufwändiger und fehleranfälliger ist. Ein Nachteil des AST-basierten Ansatzes ist, dass keine Isomorphie zwischen der Spezifikation des zu generierenden Artefaktes und dem schließlich generierten Artefakt gegeben ist (der abstrakte Syntaxbaum ist nicht unmittelbar verständlich); der Einarbeitungsaufwand in die verwendete Bibliothek zur Verwaltung und Manipulation des Objektgraphen ist unter Umständen hoch und setzt Kenntnisse im Compilerbau (oder zumindest ein gewisses Verständnis für Parser-Konzepte) voraus. Die AST-Bibliothek muss notwendigerweise die gesamte Zielsprache abdecken, selbst wenn der konkrete Anwendungsfall nur eine Untermenge davon nutzt; diese Komplexität muss vom Entwickler mitgetragen werden, was eine kognitive Belastung darstellt. Das AUTOSAR-Werkzeug Tresos Studio von Elektrobit erlaubt diesen Ansatz (zusätzlich zu dem auch dort vorwiegend verwendeten Template-basierten Mechanismus) in seinem „C Data Structures Generator“, allerdings nur für die Untermenge von C, die Variablen- und Strukturdefinitionen enthält [Ele].

Es gibt verschiedene Aspekte, die die Einführung einer Template-Sprache in den Generierungs-Prozess motivieren und rechtfertigen:

- Die abstrakte Definition der Ausgabe und die konkrete Form der Ausgabe sollen möglichst isomorph gehalten werden, so dass bei Änderungen des Ausgabeformates diese Änderungen leicht auf die generischen Templates übertragen werden können. Unter Umständen kann dadurch die Möglichkeit offen gehalten werden, die Templates durch den Benutzer und nicht nur durch den Tool-Hersteller veränderbar zu machen, was in manchen Fällen eine zusätzliche Dimension der Variabilität ermöglicht. Benutzer können so neue Anwendungsfälle abdecken, es besteht jedoch die reale Gefahr, dass Lösungen, die auf derartigen Änderungen aufbauen, kaum mehr durch den Tool-Hersteller wartbar sind. Die Voraussetzung dafür ist, dass die Templates dynamisch interpretiert werden, so dass sich Änderungen ohne ein Neuübersetzen des Code-Generierungs-Frameworks auswirken können. Ein Vorübersetzen der Templates (z.B. durch das Generieren von Java-Bytecode durch einen speziellen Template-Compiler), wie es von manchen Frameworks unterstützt wird, steht diesem Aspekt unter Umständen entgegen.

- Die Definition der Ausgabe soll möglichst entkoppelt sein von der Datenaufbereitung, d.h. die Templates sollen frei sein von jeglicher Logik,

die nicht unmittelbar das Format der Ausgabe betrifft. Kenntnisse in der Treiber-Sprache, d.h. der Host-Sprache, die die Code-Generierung steuert, sollen möglichst nicht notwendig sein, um eine Anpassung der Templates vorzunehmen. Im Umkehrschluss bedeutet dies, dass die Templates mit vollständig aufbereiteten, konsistenten Daten parametrisiert werden müssen.

- Es sollen verschiedene Zielformate unterstützt werden, d.h. entweder verschiedene Zielsprachen oder Varianten in derselben Zielsprache, die unterschiedlich genug sind, um eigene Templates zu verwenden anstatt die Variabilität als Fallunterscheidungen in bestehenden Templates zu integrieren (z.B. durch Präprozessor-Direktiven).

3.1.4 Klassifikation von generierten Artefakten

Im Hinblick auf die Tatsache, dass bestimmte Teile einer generierten Restbussimulation durch den Benutzer erweiterbar sein müssen, ergibt sich die Problematik, dass es in einem derartigen Projekt drei Kategorien von Quellcode gibt:

Rein generiert Quellcode wird generiert, eine Erweiterung oder Modifikation durch den Benutzer ist jedoch nicht zulässig.

Reiner Benutzer-Code Quellcode liegt vollständig unter der Kontrolle des Benutzers.

Generiert und durch den Benutzer erweitert Quellcode wird generiert und kann durch den Benutzer in bestimmter, definierter Weise[8] ergänzt oder modifiziert werden. Entscheidend ist hier die Frage, was mit diesem Quellcode passieren soll, wenn neu generiert werden muss.

Ziel ist es, die dritte Kategorie zu vermeiden bzw. die Zahl der Quellcode-Artefakte und den Umfang der Quellcode-Anteile, die in diese Kategorie fallen, möglichst gering zu halten. Die Diskrepanz zwischen diesem Ziel und der Notwendigkeit, dennoch einen Mechanismus bereitzustellen, um generierten Code und benutzerdefinierten Code zu verknüpfen, wird in der Literatur manchmal als *Generation Gap* bezeichnet [Vli].

Dabei geht es um zwei zusammenhängende Aspekte:

[8] Die Art und Weise, wie Erweiterungen möglich sind, kann hier entweder durch Konventionen gefordert werden, z.B. beschrieben durch eine wie auch immer geartete Benutzerdokumentation, oder durch statische Prüfungen validiert werden, falls die erlaubten Erweiterungen durch die Semantik und das Typsystem der Zielsprache prüfbar sind (d.h. durch den Compiler für die Zielsprache).

- Wie werden Benutzererweiterungen mit dem Framework-Code verknüpft, d.h. wie weiß das Framework, welche Erweiterungen durch den Benutzer implementiert wurden?
- Wie werden Modifikationen durch den Benutzer übernommen, wenn neu generiert werden muss?

Beim ersten Aspekt gibt es, abhängig von der Zielsprache, bewährte Mechanismen, wie die Anbindung von Framework-Code und benutzerdefiniertem Code erfolgen kann. Ist die Zielsprache objektorientiert und erlaubt sie die Definition von Klassen mit abstrakten Methoden, so können Erweiterungen durch den Benutzer derart erlaubt werden, dass für einen bestimmten zu erweiternden Aspekt eine abstrakte Basis-Klasse (mit leeren oder abstrakten Implementierungen) generiert wird. Der Benutzer muss bestimmte Methoden dieser Basis-Klasse überschrieben, indem er eine eigene Implementierung dieser Basis-Klasse definiert und eigene Implementierungen für bestimmte Methoden bereitstellt. Die Anbindung sämtlicher Funktionalität für diesen Aspekt erfolgt, indem die konkrete Klasse statt der Basis-Klasse vom Framework verwendet wird. Nicht erweiterte Funktionalität ist unproblematisch, da die leere Implementierung der Basis-Klasse verwendet wird. Erweiterungen, für die eine Implementierung erzwungen werden soll, können in der Basis-Klasse als rein abstrakte Methoden definiert werden, so dass die Implementierung durch den Benutzer zwangsläufig bei Ableitung von den Basis-Klasse erfolgen muss.

In C muss die Anbindung durch Funktionszeiger erfolgen; Erweiterungen durch den Benutzer müssen in Form neuer Funktionen implementiert werden und diese Funktionen dem Framework bekannt gemacht werden, indem ein bestimmter Funktionszeiger des Frameworks auf die implementierte Funktion gesetzt wird. Das Standard-Verhalten bei fehlenden Erweiterungen durch den Benutzer kann entweder realisiert werden, indem die zu setzenden Funktionszeiger auf eine leere Funktion zeigen oder indem die Funktionszeiger mit `NULL` initialisiert werden. Letzterer Fall ist sinnvoll, um unnötige Aufrufe zu vermeiden, jedoch muss die Gültigkeit des Funktionszeigers an jeder Aufrufstelle überprüft werden, um Speicherzugriffsfehler zu vermeiden. Um die Granularität der Anbindung zu Erhöhen (d.h. eine Gruppe von Funktionen gesammelt anzubinden statt jede einzelne Funktion anzubinden) ist es unter Umständen sinnvoll, eine Art Callback-Interface zu definieren als `struct` von zusammengehörigen Funktionszeigern; diese Struktur kann im Benutzer-Code initialisiert und dem Framework übergeben werden, statt eine Vielzahl von Funktionszeigern übergeben zu müssen, was fehleranfälliger und weniger wartbar ist. Eine

Methode, um die Implementierung unbedingt notwendiger Erweiterungen durch den Benutzer zu erzwingen und zur Übersetzungszeit zu prüfen, gibt es streng genommen nicht[9]; der zuvor beschriebene Mechanismus des Callback-Interfaces kann jedoch auch zu diesem Zweck verwendet werden. Ein weiterer Vorteil eines Callback-Interfaces ist, dass so ein atomarer Austausch von Funktionsgruppen zur Laufzeit erleichtert wird, was z.B. die Implementierung von Zustandsautomaten erleichtert: benutzerdefinierter Code kann mehrere Instanzen eines Callback-Interfaces definierten, die jeweils auf Funktionen zeigen, die dem zu realisierenden Verhalten für einen bestimmten Systemzustand entsprechen. Beim Wechsel des Systemzustandes kann die für diesen Zustand definierte Callback-Interface-Instanz dem Framework übergeben werden.

Beim zweiten Aspekt geht es darum, dass bestimmte Teile einer generierten Anwendung nicht zur Erweiterung durch den Benutzer vorgesehen sind, wohin andere Teile dies erlauben. Bei den nicht erweiterbaren Teilen können die betroffenen Artefakte beliebig neu generiert werden, indem die alten Versionen überschrieben werden. Bei den benutzererweiterbaren Teilen ergibt sich die Problematik, dass die vom Benutzer erweiterten Artefakte nicht überschrieben werden sollten, da sonst die Erweiterungen des Benutzers verloren gehen.

Es gibt unterschiedliche Ansätze, wie dies gelöst werden kann.

- Der einfachste Weg ist es, die durch den Benutzer erweiterten Teile nie neu zu generieren. Dies entspricht dem sogenannten Scaffolding-Prinzip, d.h. es wird davon ausgegangen, dass die Generierung nur zur Erstellung eines initialen Anwendungsgerüstes genutzt wird, das einmal aus einem bestimmten Stand der Datenbasis generiert wird und danach vollständig unter der Kontrolle des Benutzers steht.

- Ein weiterer einfacher Ansatz besteht darin, die nicht als erweiterbar anzusehenden Teil neu generieren zu lassen, d.h. zu überschreiben, und die benutzererweiterbaren Teile ebenfalls neu zu generieren, aber zuvor die durch den Benutzer bereits modifizierten Artefakte zu sichern und nach dem Abschluss des Generierungs-Vorganges eine Möglichkeit zu bieten, die vorgenommenen Modifikationen wieder zu integrieren (z.B. unter Verwendung von Merge- und Diff-Werkzeugen.

- Eine Erweiterung dieses Ansatzes ist das automatische Integrieren von Modifikationen durch den Benutzer. Dies kann auf der Ebene der generier-

[9] Oder erst in der Linker-Phase: Framework-Code kann sich auf Funktionsnamen beziehen, von denen erwartet wird, dass sie in einem vom Benutzer erweiterten Modul existieren; ist dies nicht der Fall, meldet der Linker dies als nicht auflösbare Referenz.

ten Artefakte erfolgen, d.h. durch Inspektion der zuerst generierten und dann durch den Benutzer erweiterten Artefakte, Vergleich mit dem Zustand der Artefakte nach dem Neu-Generieren und textuelles Nachziehen der Änderungen (unter Verwendung von Diff- und Merge-Funktionalität), oder auf der Ebene des Datenmodells, sofern die Benutzer-Modifikationen sich darin in einer bestimmten Form manifestieren. Mit diesem Ansatz kann nicht nur das Nachziehen von Benutzer-Modifikationen realisiert werden, sondern es können zusätzlich diese Modifikationen an den neuen Stand der Datenbasis angepasst werden, sofern dies möglich ist (siehe Abschnitt 3.4.5).

3.1.5 Informationen über den Generierungs-Vorgang

Aufrufer der Generatoren müssen in der Regel über bestimmte während des Generierungs-Vorganges auftretende Ereignisse informiert werden. Typische Arten von Informationen sind generierte Artefakte, Attribute dieser Artefakte (z.B. Zeit, die zum Generieren benötigt wurde oder Größe der generierten Datei), angelegte Verzeichnisse oder verwendete Quellen.

Typischerweise werden diese Informationen genutzt, um den Generierungs-Prozess zu dokumentieren, d.h. dem Benutzer den Fortschritt anzuzeigen oder die Vorgänge in eine Datei zu loggen. Im Kontext der Eclipse-IDE werden die Informationen ebenfalls genutzt, um neu generierte Ressourcen dem Framework mitzuteilen, so dass bestimmte Projekt-Informationen aktualisiert werden können. So muss z.B. die CDT-Infrastruktur (Build-System und Editoren) benachrichtigt werden, wenn neue Quell- oder Header-Dateien verfügbar sind, so dass gegebenenfalls Makefiles neu generiert werden können und Scanner-Informationen (entspricht in etwa einer Datenbank zur Hilfe bei der Quellcode-Navigation) neu angelegt werden können sowie gegebenenfalls geöffnete Ansichten für Dateien, die beim Generieren gelöscht wurden, zu schließen.

Diese Informationen können grundsätzlich auf zwei Arten dem Aufrufer bereitgestellt werden: zeitnah mit dem Ereignis (Push-Ansatz, meist via Callback-Interface) oder am Ende des Generierens von Clients abgefragt werden (Pull-Ansatz, z.B. als Attribut des Generators).

Die Vorteile des Push-Ansatzes sind, dass die sukzessive gesendeten Informationen sich zur Fortschrittsanzeige eignen, was insbesondere bei sehr umfangreichen Quell- oder Ausgabe-Dateien hilfreich sein kann. Des Weiteren kann dem Aufrufer die Möglichkeit gegeben werden, den Generierungs-Vorgang dynamisch zu beeinflussen, z. B. bei bestimmten Ereignissen zu stoppen oder fehlende

Eingaben durch den Benutzer vornehmen zu lassen. Eine Variante des Callback-basierten Push-Ansatzes ist die Verwendung des generischen Observer-Patterns, z. B. durch die Verwendung des `PropertyChange`-API von Java oder (falls eine komponentenübergreifende Lösung benötigt wird) des `EventAdmin`-Services von OSGi. Hier registriert sich der Aufrufer beim Generator als `Listener` und erhält über eine generische Callback-Methode (entsprechend den Vorgaben des verwendeten API) die Ereignisse in Form von `Event`-Objekten. Vorteil und Nachteil zugleich bei dieser Variante ist die schwache Typisierung dieser `Event`-Typen, die meist kaum mehr als Container für Schlüssel-Wert-Paare sind. Dadurch können sie zwar leicht von spezifischen Generatoren um neue Attribute erweitert werden, aber die Schnittstelle ist fragil, da Aufrufer nicht wissen, welche Attribute der Generator bereitstellt. Mechanismen, um dies zu ergänzen, stehen zwar zur Verfügung, zerstören aber die Attraktivität dieser Variante (die ja gerade in der Verwendung eines generischen APIs liegt). Statt des generischen PropertChange-Listener-Mechanismus können eigene, spezifische Listener- und Event-Klassen eingeführt werden, die eine stärkere Typsicherheit aufweisen.

Der Pull-Ansatz ist einfacher zu implementieren und zu verwenden, insbesondere weil die Informationen bereits in aggregierter Form bereitgestellt werden können (ist ein Aufrufer beim Push-Ansatz nur an aggregierten Informationen interessiert, so muss er die Aggregation im Callback-Interface selbst durchführen; da dies jedoch häufig nötig ist, besteht die Gefahr, dass diese Funktionalität in mehreren Aufrufern dupliziert implementiert wird).

Eine Lösung, bei der der Aufrufer periodisch Informationen des Generators abfragt, ist nicht praktikabel, da der Synchronisationsaufwand für die Kommunikation zwischen mehreren Threads die Komplexität auf Client- und auf API-Seite unnötig erhöht. [10]

Eine der dominierenden Lösungen zur Code-Generierung, die Modeling Workflow Engine (im Rahmen des Eclipse Modeling Framework entwickelt), setzt den spezifischen Listener-Ansatz um, indem ein Interface `Issues` eingeführt wird. Eine Instanz dieser Klasse kann implementiert werden, um auf Ereignisse während aller Phasen des Generierungs-Prozesses zu reagieren und um gegebenenfalls die Ereignisse zum Zwecke einer späteren Auswertung zu aggregieren.

Eine besondere Bedeutung kommt dem Mechanismus zur Weiterleitung von Ereignissen beim Generieren im Hinblick auf Fehler zu. Der übliche Exception-Mechanismus von Java sieht keine direkte Möglichkeit vor, Exceptions systematisch zu aggregieren. Darüber hinaus eignen sich Exceptions nur zum sofortigen Ab-

[10] Durch die von Aufrufer-Seite aus ohne Lock lesbaren Collection-Klassen seit Java 1.5 sinkt dieser Aufwand allerdings.

bruch einer durchgeführten Aufgabe (fail-fast), aber im Kontext der bei der Code-Generierung häufig auftretenden langwierigen Operationen ist dieses Verhalten nicht zielführend: wenn 90 Prozent eines Vorganges erfolgreich durchlaufen wird und während des 91. Prozentes eine Ausnahmebedingung auftritt, so muss der Benutzer die Fehlermeldung interpretieren und den Prozess mit neuen Eingangsdaten neu starten. Tritt nun beim 92. Prozent ein Fehler auf, muss dieser Zyklus wiederholt werden. Zielführender und produktiver ist es, zu erlauben, den gesamten Prozess zu durchlaufen und sämtliche Fehler zu aggregieren. Der Benutzer kann im Idealfall alle Probleme beheben und den Prozess neu starten, so dass nur zwei Zyklen durchlaufen werden müssen. Gegenseitige Abhängigkeiten zwischen Fehlern (Folgefehler, Maskierung von Fehlern durch andere Fehlern) verhindern, dass dieser Idealfall immer erreichbar ist.

3.2 Phasen des Generierungsprozesses

Überträgt man die in Abschnitt 3.1.1 beschriebenen Konzepte der Trennung von Frontend und Backend sowie Entkopplung und Komplexitätsreduktion durch ein Zwischenformat auf den hier behandelten Fall der Generierung von Restbussimulationen, so ergibt sich die Anforderung, für jedes zu unterstützende Eingabeformat ein Frontend-Modul (für die Transformation des Eingabeformates in die Zwischendarstellung) sowie für jede unterstützte Zielplattform ein Backend-Modul (für die Transformation von der Zwischendarstellung in plattformspezifischen Quellcode) bereitzustellen sowie die nötige Infrastruktur und eine geeignete Zwischendarstellung zu definieren, mittels derer der benötigte Informationsfluss realisiert werden kann. Zudem ergeben sich in der Praxis zusätzliche Aspekte im Zusammenhang mit den Erweiterungen der Restbussimulation durch benutzerdefinierten Code, die in Abschnitt 3.4.5 beschrieben werden und die ebenfalls Einfluss auf die Architektur der benötigten Transformationskette haben.

Eine vollständige Transformation von Spezifikation zu übersetzbarem Quellcode setzt sich demnach aus mindestens zwei Phasen zusammen: der Transformation aus den Netzwerkbeschreibungsdateien in das intern verwendete Format und der Transformation vom internen Format in den Quelltext.

3.2.1 Dekomposition des Generierungsprozesses

Die Tatsache, dass der Generierungsprozess sich aus mehreren Stufen zusammensetzt und dass eine Systematik gegeben sein muss, die eine Verknüpfung

dieser Stufen erlaubt, wird von bestehenden Frameworks zur Code-Generierung erkannt.

Die Modeling Workflow Engine löst den Aspekt der Verknüpfung der einzelnen Stufen eines mehrstufigen Transformations- bzw. Generierungs-Prozesses durch eine Workflow-Abstraktion [Ecle], die aus einer Prozessdefinition gemäß einem XML Schema sowie einer Laufzeitumgebung zur Interpretation und Ausführung dieser Prozessdefinition besteht. Die Verknüpfung der einzelnen Stufen erfolgt, indem jedem Schritt mehrere sogenannte Eingangs- und Ausgangs-Slots zugewiesen werden können. Die Eingangs-Slots sind symbolische Referenzen auf Dateien (in der Regel serialisierte Modelle), die von dem auszuführenden Schritt benötigt werden, und die Ausgangs-Slots beschreiben, welche neuen (oder aktualisierten) Dateien nach dem Ausführen dieses Schrittes zur Verfügung stehen, auf die sich nachgelagerte Schritte beziehen können. Durch die explizite Definition der konsumierten und erzeugten Artefakte ist der Prozessablauf besser dokumentiert als bei ausschließlicher Angabe der einzelnen Schritte, wo zusätzlicher Rechercheaufwand notwendig ist, um die für jeden Schritt relevanten Dateien zu ermitteln. Üblicherweise werden die eigentlichen Zielartefakte, d.h. zu generierende Quelldateien, nicht durch die Ausgabe-Slots beschrieben; z.B. wird für einen Schritt, der eine Vielzahl von Quellcode-Dateien generiert, lediglich ein `Makefile` durch einen Ausgabe-Slot definiert, das Bezüge zu den Quelldateien aufweist, nicht jedes einzelne generierte Artefakt. Nachfolgende Schritte konsumieren dann nur das `Makefile`.

Das AUTOSAR-Werkzeug Tresos-Studio der Firma Elektrobit implementiert diesen Aspekt mittels des in der Java-Welt allgegenwärtigen Build-Tools Apache Ant [Apa], was den Vorteil hat, das die Mechanismen zur Konfiguration und Erweiterung der Prozessdefinition (d.h. der Ant-spezifischen `build.xml`-Datei) bei vielen Nutzern bereits als Vorkenntnisse vorausgesetzt werden können; eine Einarbeitung in eine spezielle Workflow-Definitions-Sprache und deren Semantik bei der konkreten Ausführung, wie dies bei Verwendung der Modeling Workflow Engine nötig ist, entfällt somit [Ele]. Als Schnittstellen zwischen den Prozessstufen ist man nicht auf die Angabe der Eingabe- und Ausgabe-Slots angewiesen, was sich bei Abhängigkeiten, die sich nicht in dieses Schema einordnen lassen, flexibler ist. Die Einheitlichkeit und Systematik des Slot-Mechanismus kann dabei selbstverständlich verloren gehen; die Ant-Build-Datei erlaubt durch ein Makro-System die Definition beliebiger Logik und ist Turing-vollständig, was dazu führt, dass die Build-Datei beliebig komplex werden kann und dies in der Regel für ein zentrales Dokument wie die Prozessdefinition nicht erwünscht ist.

Ein wesentlicher Unterschied zwischen Ant und der Modeling Workflow En-

gine besteht in der Art der Kopplung zwischen den einzelnen Schritten. Die Modeling Workflow Engine bietet zwar die Eingangs- und Ausgangs-Slots, führt aber die Schritte in der Reihenfolge aus, wie sie in der Prozessdefinition aufgeführt sind, d.h. die einzelnen Schritte sind temporal gekoppelt. Unter Ant folgt (ähnlich wie in einem `Makefile`) die Abarbeitung der Schritte gemäß den in der Prozessdefinition definierten Abhängigkeiten und sind unabhängig von der lexikalischen Reihenfolge in der Prozessdefinition. Dies führt zu besseren automatisch möglichen Optimierungen des Prozesses, ist aber für das menschliche Verständnis geringfügig komplexer als eine temporale Abhängigkeit.

3.2.2 Dekomposition der einzelnen Schritte

Zusätzlich zur Dekomposition des Gesamt-Prozesses in einzelne Schritte ist es im Hinblick auf Wartbarkeit und Erweiterbarkeit notwendig, Mechanismen zur Dekomposition eines einzelnen Schrittes in Abhängigkeit von der Struktur der jeweiligen Eingangsdaten vorzusehen [11], d.h. eine Zuordnung von Strukturelementen zu möglichst modularen und autonomen Komponenten zu erreichen, die genau diese Strukturelemente behandeln.

Der Grundgedanke ist, dass das Eingangsmodell, das einem Schritt zu Grunde liegt, gewisse Strukturelemente enthält, die für sich gesondert betrachtet werden können, d.h. aus dem übergeordneten Kontext des Modells herausgelöst behandelt werden können (oder mit minimaler Kontext-Information behandelt werden). Für diese Strukturelemente kann die Logik zur Transformation bzw. Generierung ausgelagert werden in eigene Module.

Bei später eventuell auftretenden Erweiterungen oder Änderungen am Schema des Eingangsmodells (d.h. bei neu hinzukommenden Strukturelementen oder bei wegfallenden Strukturelementen) wird so die Anpassung des entsprechenden Generierungs- oder Transformations-Schrittes erleichtert: für neue Strukturelemente kann eine Komponente implementiert und im Haupt-Modul des Prozess-Schrittes mit dem Typ des Strukturelementes assoziiert werden; für nicht mehr zu behandelnde Strukturelemente kann die Assoziation der behandelnden Komponente mit dem betroffenen Strukturelement rückgängig gemacht werden.

Dieser Mechanismus eignet sich prinzipiell auch zur Erweiterung für benutzerspezifische Anwendungsfälle[12], so dass die Behandlung bestimmter Strukturelemente geändert werden kann, ohne die Kern-Funktionalität des übergeordneten

[11] Eine Dekomposition eines Schrittes in einzelne Schritte ist trivial durch eine sequentielle Ausführung mehrerer Schritte möglich; dies wird hier nicht betrachtet.

[12] Oder organisationsspezifische Anwendungsfälle

Prozess-Schrittes anzutasten.[13]

3.2.3 Abstraktion der Eingangsdaten

Um zu vermeiden, dass bestimmte Aspekte des Transformationsprozesses für jedes Eingangsformat individuell implementiert werden müssen, ist es sinnvoll, die gemeinsamen Aspekte der unterschiedlichen Eingangsformate so zu abstrahieren, dass diese teilweise Duplikation von Lösungen nicht mehr nötig ist. Das so definierte Zwischenformat blendet notwendigerweise manche Informationen aus, die in den individuellen Eingangsformaten zur Verfügung stehen; ist dies nicht akzeptabel, so muss ein Mechanismus geschaffen werden oder nachrüstbar sein, wie diese Abstraktionsschicht bei Bedarf umgangen werden kann und direkt auf die Eingangsdaten zugegriffen werden kann; dies wird meist mit dem Begriff Traceability bezeichnet. Dies betrifft insbesondere die in DBC-Dateien möglichen benutzerdefinierten Attribute, die ein eigenes Metamodell zur Beschreibung beliebiger Attribute für Netzwerkelemente und Zuordnung beliebiger Werte zu diesen Attributen bilden. Dadurch ist es möglich, dass jeder Hersteller für bestimmte Aspekte des Kommunikationsnetzwerks eine eigene Semantik festlegen kann, die beispielsweise das Verhalten von Knoten zur Laufzeit der Simulation beeinflusst. Andere Eingangsformate, wie Fibex, unterstützen ein derartiges benutzerdefiniertes Metamodell für Attribute nicht; aus diesem Grunde ist es nicht sinnvoll, diesen Mechanismus eines einzelnen Eingangsformates in der Abstraktionsschicht nachzubilden.

Ein zusätzlicher Aspekt, der durch das Zwischenformat abgedeckt werden muss, ist die Verwaltung von Informationen, die nicht Teil der Eingangsdateien sind, sondern Teil der vom Benutzer definierten Restbussimulation. Diese Informationen umfassen mindestens die folgenden Aspekte:

Aktivierung von Knoten und Botschaften Es muss beschrieben werden können, ob bestimmte Netzwerkelemente Teil der simulierten Elemente sind (d.h. ob diese von der Restbussimulation simuliert werden sollen) oder ob sie als real vorhanden angenommen werden sollen.

Callbacks Es muss beschrieben werden können, ob es für ein bestimmtes Ereignis eine vom Benutzer angegebene Callback-Funktion gibt, die vom Framework beim Eintreten des Ereignisses aufgerufen werden muss.

[13] Im Idealfall liegen die Strukturelemente im Eingangsformat so vor, dass bei Verwendung eines Template-Ansatzes das Element direkt, d.h. ohne weitere Aufbereitung, an den zugehörigen Template-Satz weitergegeben werden kann, aber dies ist eher die Ausnahme.

Weitere Informationen, z.B. über das Projekt-Setup und die Zuordnung von physikalischen Kanälen zu Controllern der Ziel-Hardware und die Parameter-Belegung dieser Controller, könnten ebenfalls in diese Abstraktionsschicht übernommen werden, können auf Grund des fehlenden Bezugs zu den Eingangsdaten aber auch extern verwaltet werden.

Für die Definition der Callbacks ist relevant, dass sich das Callback nicht auf ein bestimmtes Element bezieht, sondern auf eine bestimmte Beziehung eines Elementes zu einem anderen Element, d.h. für die vollständige Beschreibung der Restbussimulation müssen die Beziehungen zwischen Elementen u.U. expliziter gemacht werden, als diese im Informationsmodell des Eingabeformates vorhanden sind. Anders formuliert: das Eingabeformat beschreibt das Kommunikationsnetzwerk als Graph zwischen den Elementen Knoten, Nachrichten und Signalen; für die Konfiguration einer Restbussimulation sind die Kanten dieses Graphen jedoch genauso wichtig wie die Elemente selbst und müssen deshalb im Zwischenformat explizit gemacht werden.

Aus diesem Grund erleichtert es die Code-Generierung, das Eingabemodell in dieser Beziehung zu expandieren, d.h. die Abstraktionsschicht muss manche im Eingangsformat vorhandene implizite Beziehungen als vollwertige Elemente aufnehmen. Diese teilweise Linearisierung der Eingabedaten ermöglicht eine Vereinfachung des Code-Generierungs-Schrittes, da für ein zu behandelndes Objekt weniger Kontext-Informationen mit übergeben werden müssen bzw. diese Kontext-Informationen nicht während des Generierungs-Schrittes berechnet werden müssen. Speziell zu diesem Aspekt gibt es interessante Unterschiede zwischen dem DBC-Format, das Beziehungen sehr redundanzfrei modelliert und somit die Freistellung bestimmter Beziehungen erschwert, und neueren Formaten wie Fibex und AUTOSAR-System-Konfiguration, in denen viele Beziehungen sowohl auf System-Ebene als auch auf der Ebene der einzelnen ECU beschrieben sind, was eine gewisse Redundanz bei der Modellierung erfordert, aber die Extraktion bestimmter Beziehungen erleichtert.[14]

Eine weitere Aufgabe, die mittels der Zwischenschicht realisiert werden kann, ist das dokumentübergreifende Validieren der Datenbasis, was wichtig ist für den Anwendungsfall, dass mehrere Eingangsdateien einem physikalischem Kanal zugeordnet werden sollen, um die mehrfache Verwendung gleicher Botschafts-Identifier zu vermeiden bzw. den Benutzer darüber zu informieren.

Auch die in Abschnitt 3.4.5 beschriebene Funktionalität ist ohne Verwendung einer einheitlichen Abstraktionsschicht nur durch aufwändiges Ermitteln von

[14] Dies impliziert nicht, dass Fibex- oder AUTOSAR-Beschreibungen generell einfacher zu behandeln sind; dies ist auf Grund der komplexeren Strukturen nicht der Fall.

Querbezügen und evtl. durch das dokumentübergreifende Auflösen von Beziehungen realisierbar. Die Ermittlung der Differenz zwischen zwei verschiedenen Versions-Ständen wird einerseits erleichtert durch die Komplexitätsreduktion, die die Abstraktionsschicht schafft (d.h. nicht relevante Aspekte werden automatisch ausgeblendet). Andererseits muss durch Verwendung eines vereinheitlichten Formates für die Zwischendarstellung der Algorithmus zur Bildung der Differenz nur für dieses Format implementiert werden und nicht für jedes einzelne Eingabeformat (bzw. im Extremfall für jede Kombination von Eingabeformaten).

3.2.4 Transformation Eingangsdaten → Zwischenformat

Die Transformation der Eingangsdaten in das Zwischenformat muss weiter unterteilt werden in mehrere Schritte: das Einlesen der Eingangsdaten aus der serialisierten Darstellung (d.h. aus Dateien), die eigentliche Transformation der Daten in das Zwischenformat, die Zusammenführung von Daten aus mehreren Quellen und die damit verbundene Konflikterkennung sowie die Serialisierung des Zwischenformates.

Die ersten beiden Schritte sind spezifisch für den Typ der verwendeten Eingangsdaten, während die übrigen Schritte generisch sein können, da sie nur auf der Zwischendarstellung operieren.

Der erste spezifische Schritt, das Einlesen der serialisierten Daten, erfolgt ja nach verwendetem Datenformat entweder durch geeignete speziell zu entwickelnde Parser; so stehen für DBC-Dateien und LDF-Dateien von Berger Elektronik entwickelte Parser zur Verfügung [Sar11, Hit10, Fra10]. Im Falle von XML werden üblicherweise XML-Parser oder XML-Mapper wie JAXB oder EMF verwendet.

Die verwendete Technologie zum Einlesen der Daten beeinflusst auch die zur Verfügung stehenden Optionen für die Realisierung des Transformations-Schrittes.

In jedem Fall möglich ist eine manuelle Implementierung der Transformation, d.h. die Überführung des Datenmodells, wie es der Parser liefert, in das Zwischenformat mit den Mitteln der Host-Sprache. Im Wesentlichen muss hierzu das Datenmodell traversiert werden und zu Elementen im Eingangsmodell die korrespondierenden Elemente im Zwischenformat erzeugt und parametriert werden. Bei speziellen Parsern, die die Daten über ein individuelles Datenmodell zugänglich machen, ist dies oft der einzige gangbare Weg. Entwurfsmuster wie Visitor und Builder haben sich zur Strukturierung dieser Lösungsvariante bewährt [GHJ94].

Existiert das Eingangsformat als XML- oder EMF-Modell, und ist das Zwischenformat ebenfalls als (anderes) XML- oder EMF-Modell definiert (wobei die Modelle jeweils dem XSD-Metamodell bzw. dem Ecore-Metamodell entsprechen), so kann der Einsatz von speziellen Mapping-Technologien in Betracht gezogen werden. Diese ermöglichen es, die Transformation deklarativ zu definieren, was insbesondere bei größeren Modellen den Aufwand gegenüber einer manuellen Implementierung reduzieren kann. Der Nachteil derartiger Transformations-Sprachen ist, dass u.U. ein hoher Einarbeitungsaufwand nötig sowie spezielle Werkzeugunterstützung (Editoren und Compiler für die Transformationssprache) notwendig.

Einen breiten Überblick über Sprachen zur Modelltransformation sowie zur Theorie der Triple-Graph-Grammatiken, die vielen dieser Sprachen zu Grunde liegt, bietet [Hub08]. Eine genauere Beschreibung der Theorie der Triple-Graph-Grammatiken liefert [KW07].

In der vorliegenden Arbeit werden explizite Modelltransformations-Sprachen nicht weiter betrachtet; durch die Implementierung der Zwischenschicht mittels EMF ist jedoch eine zukünftige Transformation von anderen EMF-Modellen, z.B. mittels der Transformations-Sprache ATL denkbar [Ecla].

3.2.5 Transformation Zwischenformat → Ausgabeformat

Die Transformation vom Zwischenformat in das Ausgabeformat ist die eigentliche Code-Generierung. Fast alle industriell eingesetzten Lösungen hierfür sind als duale Ansätze realisiert, bei denen ein Treiber in einer Host-Sprache und eine Beschreibung der Ausgabe in einer Template-Sprache kombiniert werden (siehe Abschnitt 3.1.3).

3.3 Projekt-Modell

Zur Zusammenstellung der zu einer zu generierenden Restbussimulation gehörigen Informationen wurde eine Projekt-Abstraktions-Schicht geschaffen, die die für das Projekt relevanten Datenbasen, die Zielplattform für das Projekt, die vorhandenen physikalischen Kanäle (Art, Bezeichnung und Parameter) und die Zuordnung der Datenbasen zu den Kanälen einerseits sowie die Zuordnung der Kanäle zu den Anschlüssen der Zielplattform andererseits ermöglicht. Die Zuordnungen zwischen `ICluster`, `IChannel` und `IConnector` orientieren sich im Kern an der vom Fibex-Standard definierten Topologie (die von AUTOSAR weitgehend übernommen wurde). Diese Informationen werden als Projekt-Konfigurations-

Datei gespeichert und dienen als Treiber für die für das Projekt verwendeten Code-Generatoren. Eine Übersicht über die Beziehungen ist in Abbildung 3.1 zu finden.

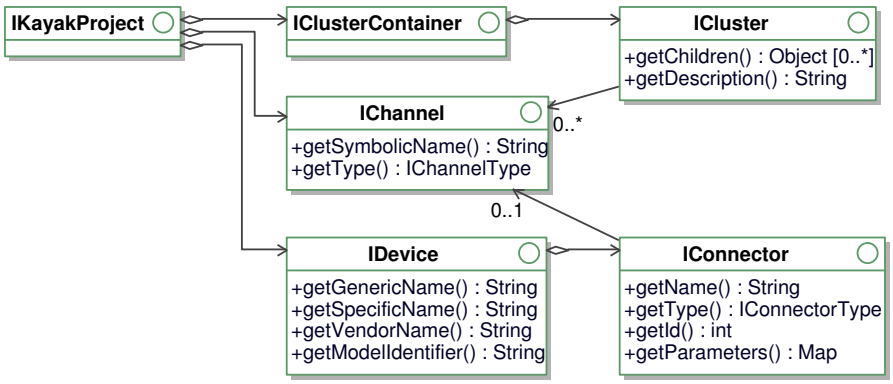

Abbildung 3.1: Abstrakte Projekt-Struktur [Ber]

Im Folgenden werden die Elemente der Projekt-Konfiguration näher beschrieben.

3.3.1 Abstraktion der Datenbasen

Da mehrere unterschiedliche Eingangs-Formate unterstützt werden müssen, ist eine Abstraktionsschicht für die Einbindung dieser Formate notwendig, die das Referenzieren konkreter Dateien und die Deserialisierung dieser Daten mittels des für das spezifische Datenformat verwendeten API ermöglicht (z.B. Verwendung generischer XML-APIs oder eines spezifischen Parsers für spezielle Datenformate), ohne dass die Projekt-Abstraktions-Schicht diese APIs kennen muss. So kann gewährleistet werden, dass für weitere zu unterstützende Datenformate keine Änderungen an der Projekt-Abstraktions-Schicht notwendig sind, sondern diese als externe Funktionalität eingebunden werden können (wie dies konkret funktioniert, d.h. wie die Anbindung externer Funktionalität realisiert wird, ist in Abschnitt 3.5 beschrieben). Die Einführung von `IClusterContainer` als übergeordnete Instanz über `ICluster` motiviert sich aus der Tatsache, dass manche Eingangsformate die Definition mehrerer Netzwerk-Cluster innerhalb einer Datei ermöglichen, so dass `IClusterContainer` sich auf die Ressource bezieht, die die Netzwerk-Cluster definiert, während sich `ICluster` auf einen einzelnen

Netzwerk-Cluster bezieht und die generische Operation `getChildren` bereitstellt, die in abstrakter, vom konkreten Eingangsformat unabhängiger Weise den Inhalt der Clusters zugänglich macht.

3.3.2 Abstraktion der physikalischen Kanäle

Die physikalischen Kanäle, die für eine zu generierende Restbussimulation relevant sind, werden beschrieben durch den Bus-Typ des physikalischen Busses und einen Parametersatz zur Beschreibung der Parameter für den konkreten Bus; dieser Parametersatz ist auf der Ebene der Projekt-Konfiguration abstrakt gehalten. Wichtig für die Restbussimulation ist insbesondere der symbolische Name eines physikalischen Busses, da dieser sich auswirkt auf die generierten API-Funktionen für die Restbussimulation.

3.3.3 Abstraktion der Zielplattform

Die Zielplattform wird dem Projekt explizit zugewiesen. Sie ist definiert durch eine Reihe beschreibender Merkmale sowie den ihr zugeordneten Anschlüssen, für die der Anschluss-Typ und mögliche Parameter definiert werden können. Die Anschlüsse dienen als Endpunkte für physikalische Kanäle. Die Menge der unterstützten Zielplattformen ist nach oben hin offen, so dass neue Plattformen (oder Plattform-Varianten) als externe Module eingeführt werden können, ohne das Kern-System zu verändern (siehe Abschnitt 3.5).

Das Anwendungsprofil ist ein zusätzliches Merkmal, anhand dessen die Klasse der zu generierenden Anwendung definiert wird. Dies dient im Wesentlichen dazu, unterschiedliche typische Anwendungsarten wie Gateways oder Logger für eine Plattform generieren zu können, ohne dass diese Anwendungstypen als eigene Zielplattformen definiert werden müssen. Auch kundenspezifische Anwendungstypen können so definiert werden.

Die Auswahl der benötigten Generatoren für ein Projekt erfolgt dann mittels der Kombination aus Zielplattform und Anwendungsprofil (auch hierzu sei auf Abschnitt 3.5 verwiesen).

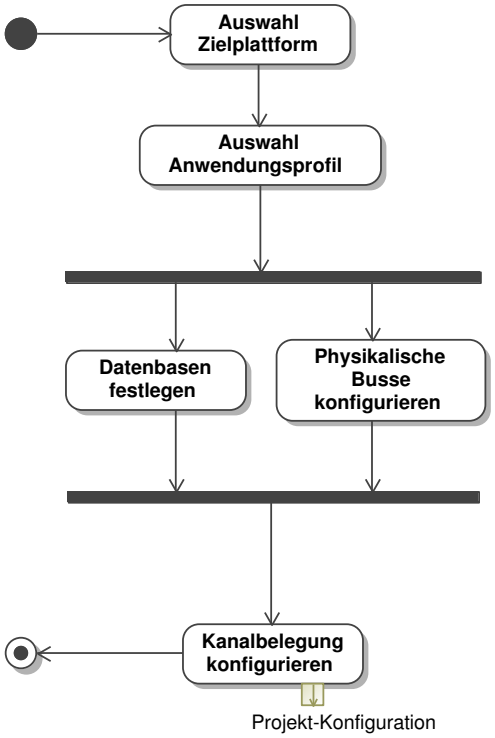

Abbildung 3.2: Ablauf der Projekt-Konfiguration [Ber]

Der Ablauf der Projekt-Konfiguration ist in Abbildung 3.2 dargestellt. Eine kurze Erläuterung: die Zielplattform und das Anwendungsprofil werden ausgewählt. Die zu verwendenden Datenbasen werden eingebunden. Die physikalischen Kanäle werden definiert und die Datenbasen werden diesen zugewiesen. Abschließend werden die physikalischen Kanäle den Anschlüssen der Zielplattform zugeordnet.

Ein konkretes Beispiel für eine Projekt-Konfiguration, die die Zuordnung zwischen den Projekt-Elementen und der zugehörigen realen Topologie zeigt, ist in Abbildung 3.3 dargestellt. Hier ist die Projekt-Struktur für eine Konfiguration dargestellt, die für eine Topologie mit zwei physikalischen Kanälen (Chassis und Powertrain) jedem Kanal die zugehörige Datenbank zuordnet sowie die Anbindung der Anschlüsse der verwendeten Zielplattform an diese Kanäle definiert.

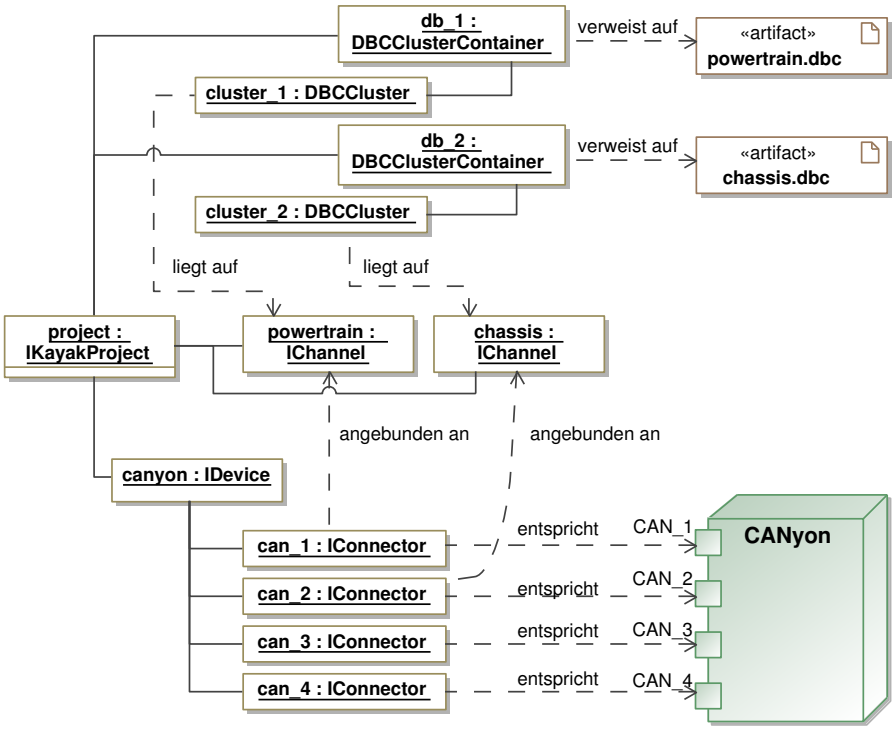

Abbildung 3.3: Konkretes Beispiel für eine Projekt-Konfiguration [Ber]

3.4 Generator-Architektur

Im Folgenden wird eine Umsetzung der in Abschnitt 3.2 beschriebenen Anforderungen vorgestellt. Wo möglich, werden Einzelheiten an Hand von konkreten Beispielen veranschaulicht.

3.4.1 Kern-Abstraktionen und deren Interaktionen

Den Kern des entwickelten Generator-Frameworks bilden die Abstraktionen `IGenerator` und `IComponentContribution`.

`IGenerator` ist die Schnittstelle für einen einzelnen Schritt im Generierungs-Prozess. Eine konkrete Implementierung dieser Schnittstelle realisiert die in diesem Schritt auszuführenden Transformationen. Die inhaltsbasierte Behandlung

von Eingangsdaten erfolgt über einzelne Implementierungen der Schnittstelle `IComponentContribution`, die mittels der Schnittstelle `IContributionHandler` an eine `IGenerator`-Instanz gebunden werden können. Dadurch wird eine Zuordnung zwischen Strukturelementen zu ihren behandelnden Modulen erreicht.

Eine `IGenerator`-Instanz ruft die ihr zugeordneten `IComponentContribution`-Instanzen auf, indem sie über ihre Eingangsdaten iteriert (wie dies erfolgt, ist abhängig von den verwendeten Eingangsdaten für den von diesem Generator implementierten Schritt) und für jedes potentiell behandelbare Strukturelement überprüft, ob eine der `IComponentContribution`-Instanzen dieses Element behandeln kann. Diese Prüfung erfolgt mittels der Methode `isContextSupported` von `IComponentContribution`. Ist diese Überprüfung positiv, so wird die `contribute`-Methode dieser `IComponentContribution`-Instanz aufgerufen, indem deren `contribute`-Methode aufgerufen wird, wobei das zu behandelnde Element nicht direkt übergeben wird, sondern verpackt in eine `IContext`-Instanz, die zusätzlich nötige Informationen über den aufrufenden Generator weitergibt.

Die so aufgerufene `IComponentContribution`-Instanz transformiert das ihr übergebene Element, indem sie es, je nach konkretem Fall, in ein Element des Zielmodells überführt oder indem sie Templates für dieses Element instantiiert und das Element an diese Templates übergibt, in der Regel mit zusätzlichen Parametern. Dabei erfolgt die Ausgabe in Form eines finalen, als Datei generierten Artefaktes nicht direkt, sondern es wird ein Event vom Typ `IArtifactEvent` erzeugt und über die `IArtefactHandler`-Schnittstelle an die aufrufende `IGenerator`-Instanz übergeben. Die Motivation für diese Indirektion erfolgt im nächsten Abschnitt. Ebenso ist es möglich, dass eine `IComponentContribution`-Instanz einen Beitrag zu Artefakten leisten will, die von einer anderen `IComponentContribution`-Instanz generiert werden. Dies ist so möglich, dass die Intention dieses Beitrags in Form einer `IAttributeEvent`-Instanz über die Schnittstelle `IAttributeHandler` an den aufrufenden Generator weitergeleitet wird.

Eine Übersicht über die beteiligten Kern-Abstraktionen des Generator-Frameworks bietet Abbildung 3.4.

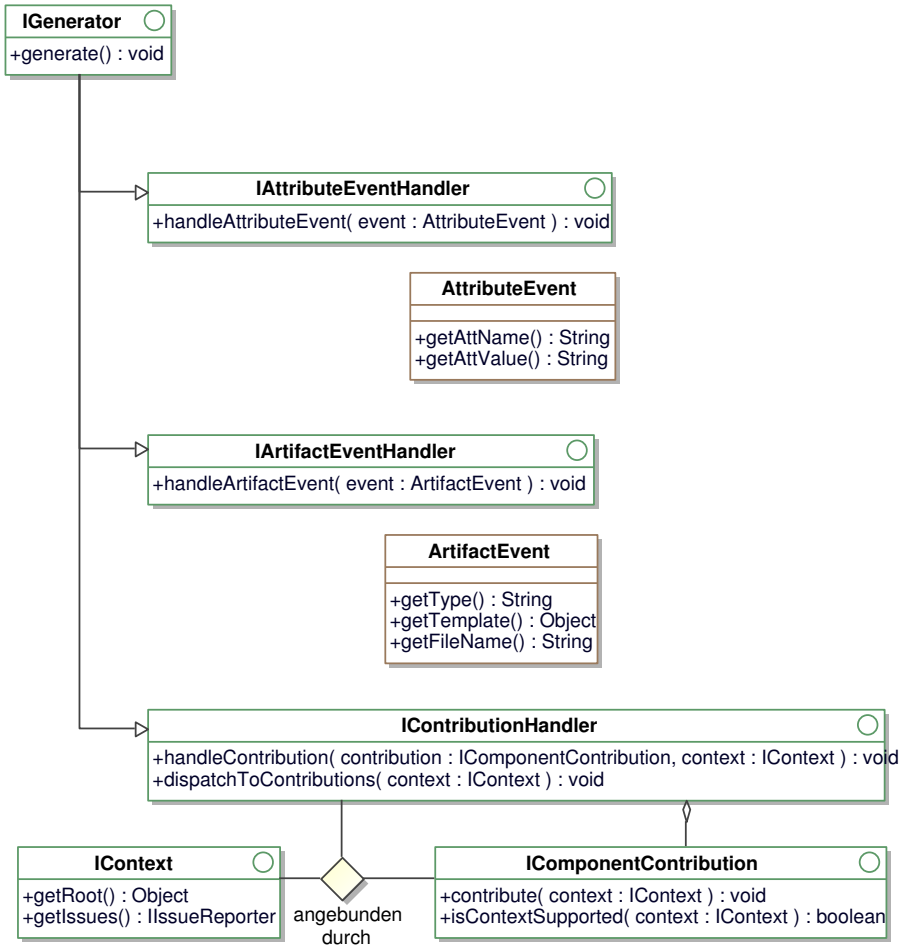

Abbildung 3.4: Kern-Abstraktionen des Generator-Frameworks [Ber]

Um die Realisierung von `IGenerator`-Implementierungen zu vereinfachen und zu vereinheitlichen, wird die Klasse `AbstractGenerator` als abstrakte Basis-

Klasse bereitgestellt, die die unter anderem bereits die Infrastruktur für die Registrierung von `IComponentContribution`-Instanzen bereitstellt.

Die weitere Unterteilung der Generator-Funktionalität, d.h. die konkreten Unterklassen von `IGenerator`, fällt in zwei dominierende Kategorien:

C-Code-Generierung Für Generatoren, die die eigentliche Generierung von C-Code durchführen, wird als spezifische Basis-Klasse die Klasse `BaseCCodeGenerator` bereitgestellt, die eine Visitor-Methode implementiert, die über die Strukturelemente des Zwischenformates iteriert und die auf diese Strukturelemente registrierten `IComponentContribution`-Instanzen aufruft.

Modell-Transformation Für Generatoren, die eine Transformation der Eingangsformate in das Zwischenformat oder eine Vorkonditionierung oder Konsolidierung des Zwischenformates realisieren, wird die Basis-Klasse `AbstractModelTransformer` bereitgestellt, die einfache Methoden zum Serialisieren und Deserialisieren des Zwischenformates und die Ermittlung der Differenzen zwischen zwei Instanzen des Zwischenformates realisiert.

Diese beiden Basis-Klassen bilden ein Vokabular, mit dem die benötigte Funktionalität für die Transformations-Kette in konsistenter Weise abgebildet werden kann, indem die Funktionalität jedes konkreten Prozess-Schrittes als Unterklasse einer dieser Basis-Klassen erfolgen kann. Für Prozess-Schritte, die sich nicht in eine dieser Kategorien einordnen lassen, kann eine Implementierung erfolgen, die direkt auf `IGenerator` bzw. `AbstractGenerator` aufsetzt und die beiden bevorzugten Basis-Klassen können ignoriert werden. Die wichtigsten konkret verwendeten Unterklassen sind in Abbildung 3.5 dargestellt; wichtig ist, dass alle plattformspezifischen Aspekte in einem plattformspezifischen Generator gekapselt sind und alle übrigen Generatoren für alle Plattformen genutzt werden können.

Als bevorzugte Basis-Implementierung für `IComponentContribution` existiert die Klasse `AbstractComponentContribution`, die bevorzugt zu verwenden ist und die die notwendigen Anbindungen dieser Schnittstelle (Anbindung an `IArtifactHandler` und `IAttributeHandler`) in einheitlicher Weise realisiert und einfache Methoden bereitstellt, um `IArtifactEvent`- und `IAttributeEvent`-Instanzen zu erzeugen und an die entsprechenden registrierten Handler (dies sind in der Regel die aufrufenden `IGenerator`-Instanzen) zu propagieren.

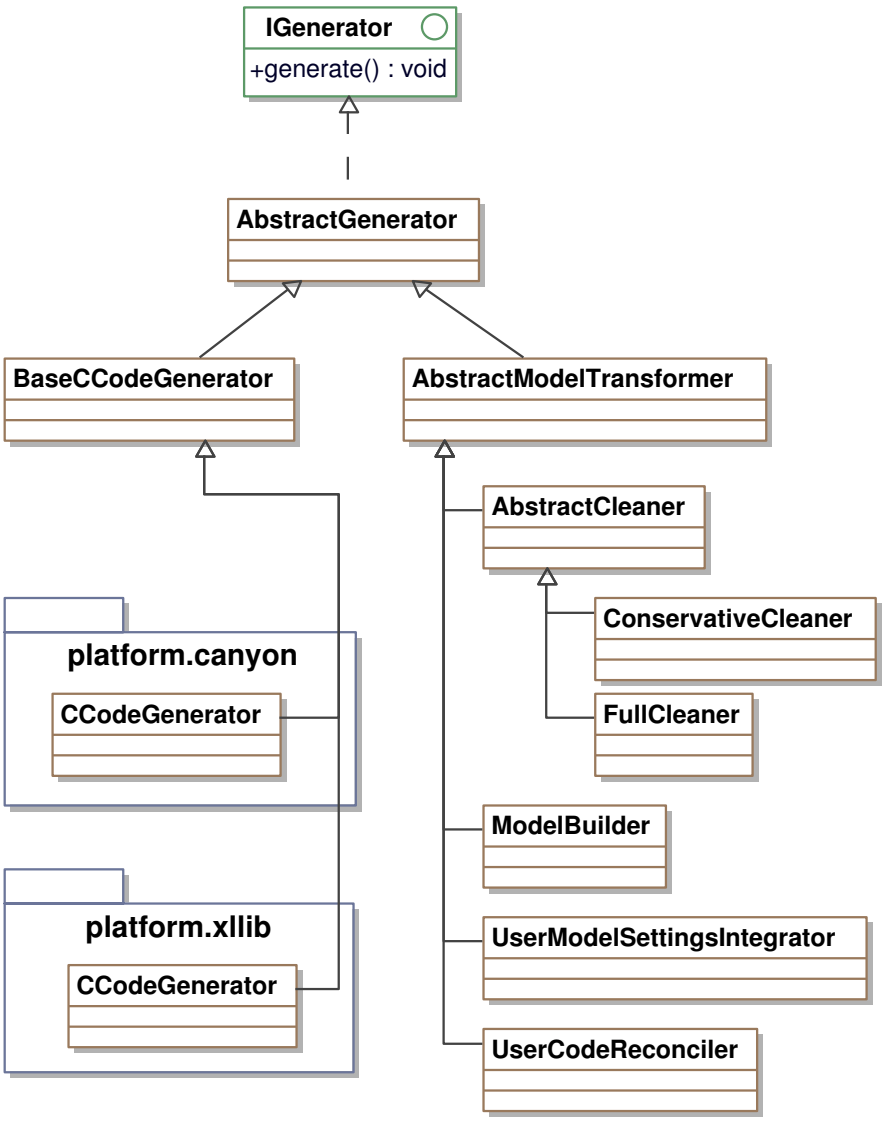

Abbildung 3.5: Verwendete Generatoren und plattformspezifische Anteile [Ber]

3.4.2 Behandlung von Artefakt- und Attribut-Beiträgen

Die zuvor erwähnte Schnittstelle `IArtifactHandler`, die von jeder `IGenerator`-Unterklasse implementiert wird, ermöglicht es, Artefakte nicht direkt zu generieren, sondern detailliert zu steuern, wie und zu welchem Zeitpunkt das von einem `IArtifactEvent` spezifizierte Artefakt generiert wird. Die Motivation dafür ist, dass es durch die Möglichkeit der Erweiterung der Restbussimulation durch den Benutzer zwei Klassen von zu generierenden Artefakten gibt: bestimmte Quelldateien sind nicht für die Erweiterung durch den Benutzer vorgesehen, d.h. sie werden nur generiert, aber nichts an diesem generierten Quellcode darf vom Benutzer verändert werden. Diese Artefakte können somit immer wieder neu generiert werden, ohne dass darauf geachtet werden muss, ob sie Änderungen durch den Benutzer enthalten. Artefakte, die der Klasse der durch den Benutzer erweiterbaren Dateien angehören, dürfen nicht in jedem Fall überschrieben werden, sondern müssen entweder belassen werden, wie sie sind, oder es muss, wenn eine solche Datei neu generiert werden soll, die alte Version gesichert werden, bevor sie überschrieben wird.[15]

Der Typ des Artefaktes wird dabei über ein Feld des entsprechenden `IArtifactEvents` bestimmt, das von der Ereignis-Quelle, d.h. der erzeugenden `IComponentContribution`-Instanz, entsprechend korrekt gesetzt werden muss. Mit dieser Information kann der Generator zentralisiert entscheiden, wie das behandelte Artefakt erstellt werden muss, indem es je nach Typ die entsprechende Strategie auswählt.

Ein weiterer Vorteil der Kapselung von Informationen zu jedem zu generierenden Artefakt ist, dass eine Erweiterung auf andere unterstützte Template-Arten dadurch an einer zentralen Stelle erfolgen kann, indem ein zusätzlicher Handler für Artefakt-Ereignisse mit der neuen Template-Variante implementiert und registriert wird. Aspekte wie die Gewährleistung der Atomizität der Erstellung von Artefakten, d.h. die Sicherheit, dass ein Artefakt entweder vollständig und korrekt generiert wird oder das ursprüngliche Artefakt an seiner Stelle verbleibt, können ebenfalls als zentrale Strategie für manche Artefakt-Typen eingebunden werden.

Im Hinblick auf die Generierung von C-Code ist in diesem Zusammenhang eine Optimierung möglich, indem ein neu zu generierendes Artefakt nur als Dateisystem-Datei geschrieben wird, wenn der Inhalt sich vom bisher an dieser Stelle befindenden Artefakt unterscheidet (dies kann z.B. mittels einer kryptographischen Prüfsumme überprüft werden); ist dies nicht der Fall, so kann

[15] Eine andere Möglichkeit der Behandlung wird in Abschnitt 3.4.5 beschrieben.

das ursprüngliche Artefakt verbleiben, und der `make`-Prozess kompiliert die entsprechende Quelldatei (und evtl. davon abhängige Dateien) nicht neu, was bei größeren Projekten eine für den Benutzer signifikante Zeitersparnis bewirkt.[16]

Eine weiterer Aspekt des Generierungs-Prozesses, der durch den `IAttribute-Handler`-Mechanismus abgedeckt werden kann, ist das Löschen von Artefakten, z.B. beim Wegfall von Netzwerk-Knoten aus der Datenbasis und anschließendem Neugenerieren oder auch, um einen vollständigen Build des Projektes zu erzwingen. Auch hier kann je nach der Kategorie des zu löschenden Artefaktes zentral entschieden werden, ob das Artefakt vom Dateisystem gelöscht werden soll oder (bei Artefakten, die Änderungen durch den Benutzer enthalten können) ob das Artefakt gesichert werden muss.

Die Schnittstelle `IAttributeHandler` realisiert die Möglichkeit, dass einzelne `IComponentContribution`-Instanzen Beiträge zu Artefakten liefern, die von anderen `IComponentContribution`-Instanzen erzeugt werden. Dies nutzt die verzögerte Generierung der Artefakte aus, indem zuerst alle Attribut-Ereignisse abgearbeitet werden und in die entsprechenden Artefakte injiziert werden, bevor alle dem Generator mittels `IArtifactEvent` übergebenen Artefakte generiert werden. Damit ist die Reihenfolge der Ausführung, d.h. ob für ein bestimmtes Artefakt zuerst die Attribut-Ereignisse oder das Artefakt-Ereignis an den Generator geleitet werden, nicht relevant, was der Tatsache Rechnung trägt, dass die einzelnen `IComponentContribution`-Instanzen möglichst nichts voneinander wissen sollen und insbesondere nicht in temporaler Abhängigkeit voneinander stehen sollen. Die Assoziation zwischen dem Eigner des Artefaktes, für das ein Attribut gesetzt werden soll, und der externen Komponente, die das Attribut setzt, erfolgt ausschließlich über den Namen des Attributes, was beachtet werden muss, wenn dieser Mechanismus häufiger eingesetzt wird, da es dann u.U. zu Namenskonflikten kommen kann.

Das ereignisbasierte Behandeln der Artefakt-Generierung und Attribut-Weiterleitung hat auch Auswirkungen auf die Testbarkeit: in Testfällen können aufwändige dateisystembasierte Tests, die Prüfen, ob eine bestimmte Datei durch einen bestimmten Prozess-Schritt generiert werden, ersetzt werden durch einfache Tests, die Prüfen, ob ein entsprechendes `IArtifactEvent` für das erwartete Artefakt erzeugt wurde. Dies hilft natürlich nur bei Tests, die lediglich die Präsenz des Artefakts betreffen. Bei inhaltsbasierten Prüfungen ist der ereignisbasierte

[16] Bei Build-Systemen, die (anders als `make`), eine Neukompilierung nicht vom Zeitstempel der generierten Datei abhängig machen, sondern bereits selbst an Hand einer Prüfsumme eine inhaltsbasierte Entscheidung über eine notwendige Neukompilierung treffen, könnte dieser Schritt entfallen. Beispiele für derartige Build-Systeme sind `cons`, `SCons`, `waf`, `omake` und `fabricate`; von diesen ist `SCons` am populärsten.

Ansatz jedoch auch hilfreich, da im Kontext des Tests spezielle Artefakt-Handler verwendet werden können, die das Artefakt nicht als persistente Datei im Dateisystem erzeugen, sondern als Byte-Array im Speicher, was die Inspektion des Inhaltes erleichtert und das sonst notwendige Entfernen der während dem Test generierten Artefakte überflüssig macht. [17]

3.4.3 Fassade zur Verwendung der Generatoren

Um einerseits die richtige Sequenzierung der einzelnen für den Generierungs-Prozess verwendeten Schritte zu gewährleisten und andererseits die Integration der Code-Generierung mit dem nachgelagerten Build-Prozess, d.h. dem Kompilieren des Quellcodes und Linken des Objektcodes (realisiert durch einen Aufruf von `make`) zu ermöglichen, wurde das Interface `BuildService` eingeführt. Dieses erlaubt es, verschiedene Ausprägungen des Generierungs- und Build-Prozesses einfach auszuführen, ohne dass einzelne Prozess-Schritte konfiguriert werden müssen. Die wichtigsten dieser Ausprägungen umfassen:

Konservative Code-Generierung Es werden Änderungen aus den Eingangsdateien eingebunden und Code wird neu generiert, aber es werden dabei keine durch den Benutzer erweiterbaren Quell-Artefakte überschrieben. Typischer Anwendungsfall ist, dass ein Benutzer neue Elemente aus den Eingangsdaten einbeziehen will.

Vollständige Code-Generierung Es werden Änderungen aus den Eingangsdateien eingebunden und Code wird neu generiert und eventuell bestehende durch den Benutzer erweiterte Quell-Artefakte werden überschrieben. Typischer Anwendungsfall ist, dass ein Benutzer den ursprünglichen Zustand des erweiterbaren Quellcodes wiederherstellen kann, um so z.B. das Projekt wieder in einen kompilierbaren Zustand zu bringen. Hier ist wichtig, dass die Granularität regulierbar ist, d.h. dass die Neu-Generierung auf bestimmte Artefakte beschränkt werden kann, falls der Benutzer nur diese neu generieren will.

Nur Build Es werden keine Eingangsdaten neu eingebunden und kein Code neu generiert, sondern nur das Binary neu gelinkt und gegebenenfalls zuvor Quell-Artefakte, die durch den Benutzer verändert wurden, neu kompiliert.

Vollständiger Build Es werden Änderungen aus den Eingangsdateien eingebunden und Code wird generiert, dieser kompiliert und der resultierende

[17] Dies sind keine aus funktionaler Sicht wichtigen Aspekte, aber sie zeigen die grundsätzliche Flexibilität, die dieser Ansatz bietet.

Objektcode zu einem Binary gelinkt. Typischer Anwendungsfall ist das erstmalige Generieren, Kompilieren und Linken eines Projektes.

Die Schaffung einer Schnittstelle für die korrekte Verwendung der Generatoren isoliert die konkreten Generator-Implementierungen von Modulen, die nur ein Interesse an der korrekten Ausführung des Build-Prozesses haben und nicht an den einzelnen Schritten; dies betrifft die überwiegende Mehrheit der Module, einschließlich aller Benutzerschnittstellen.

3.4.4 Informationsfluss während des Generierungs-Vorgangs

Die in Abschnitt 3.1.5 beschriebene Thematik der Behandlung von Status- und Fehler-Meldungen während des Generierungs-Prozesses wird wie in Abbildung 3.6 dargestellt behandelt. Die Schnittstelle `IIssueReporter` ermöglicht die Realisierung des dort beschriebenen spezifischen Listener-Ansatzes. Konkrete Implementierungen realisieren die Schnittstelle in einer der Umgebung angepassten Weise; für eine Generierung außerhalb einer Entwicklungsumgebung wird die Klasse `ConsoleIssueReporter` bereitgestellt, die die vom Generierungs-Prozess gelieferten Informationen an die Standard-Ausgabe bzw. die Standard-Fehlerausgabe weiterleitet; im Kontext der Entwicklungsumgebung wird eine spezielle Klasse `UiIssueReporter` bereitgestellt, die die Umwandlung dieser Informationen in spezielle Strukturen realisiert, die in der Log-Ansicht der IDE angezeigt werden können.

Eine spezielle Methode von `IIssueReporter`, die Methode `vetoableContinuation`, erlaubt es, den Generierungs-Prozess bei bestimmten kritischen Bedingungen optional fortzusetzen oder abzubrechen. Durch die kontextspezifische Implementierung dieser Methode kann hier wiederum innerhalb der IDE eine Rückfrage an den Benutzer erfolgen und bei Verwendung außerhalb der IDE kann eine Implementierung dieser Methode so erfolgen, dass bei Fehlern der Generierungs-Prozess abgebrochen wird und bei einfachen Problemen der Generierungs-Prozess fortgesetzt wird. Prinzipiell könnte diese Behandlung durch Inspektion der Meldungen noch verfeinert werden und konfigurierbar gemacht werden.

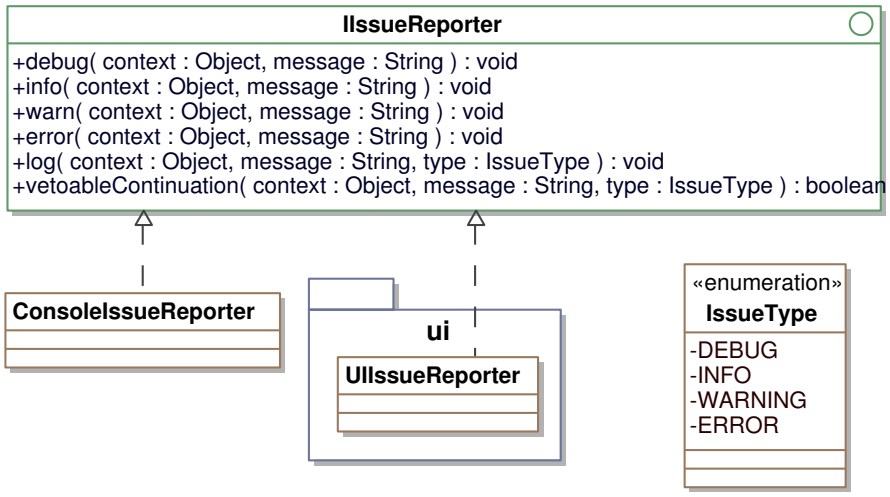

Abbildung 3.6: Verarbeitung von Statusmeldungen während der Generierung [Ber]

3.4.5 Änderungen am Modell nachziehen

Da sich die verwendete Datenbasis für die Kommunikationsmatrix in gewissem Maße ändert (entweder durch Änderungen, die von außen kommen, z.B. im Rahmen der Produktentwicklung des durch die Restbussimulation geprüften oder getesteten Produkts, oder durch anwendungsspezifische Änderungen, die der Benutzer selbst gezielt vornimmt), müssen Mechanismen integriert werden, wie die von Änderungen an der Datenbasis betroffenen Teile der Restbussimulation angepasst werden können, d.h. wie selektiv Änderungen an der Datenbasis in die bestehende Restbussimulation integriert werden können. Um die Beschreibung im Folgenden kürzer und präziser zu machen, werden die folgenden Bezeichnungen verwendet: K_{old} für den Stand der Datenbasis zum Zeitpunkt der letzten Generierung, K_{new} für den Stand der Datenbasis zu einem späteren Zeitpunkt und Δ_K für die strukturelle Differenz zwischen K_{old} und K_{new}.

Der Aspekt der Variabilität der Kommunikationsmatrix betrifft speziell den durch den Benutzer erweiterten Teil der Restbussimulation (im Folgenden mit R_{user} bezeichnet). Beim rein generierten Teil der RBS (im Folgenden mit R_{gen} bezeichnet) spielt dies keine Rolle, da dieser einfach vollständig neu generiert werden kann. Die am benutzererweiterten Teil notwendigen Anpassungen fallen in verschiedene Kategorien, die im Folgenden beschrieben werden. Die Reaktionen auf die Änderungen, d.h. wie eine Anpassung von R_{user} erfolgen muss, wird

hinter dem Symbol $\rightarrow$ angegeben (der Prozess der Anpassung von R_{user} wird im folgenden Konsolidierung genannt).

Entfernte Elemente ECUs, Botschaften oder Signale sind in K_{old} enthalten und in K_{new} nicht enthalten
$\rightarrow$ Benutzer ist verantwortlich für das Entfernen von API-Methoden, die auf diesen Elementen operieren (die betroffenen Stellen können jedoch durch das System markiert oder annotiert werden, so dass dem Benutzer ein Überblick über die durchzuführenden Änderungen gegeben wird).

Neue Elemente ECUs, Botschaften oder Signale sind in K_{new} enthalten, aber in K_{old} nicht enthalten
$\rightarrow$ Keine Konsolidierung notwendig; Benutzer kann nach dem Generieren die neuen API-Methoden in seinem bestehenden Code verwenden; eine Anpassung bestehenden Codes (automatisch oder durch den Benutzer) ist nicht erforderlich.

Änderungen an Attributen Attribut-Werte von Elementen wurden geändert; hier sind drei Fälle unterscheidbar in Bezug darauf, wie diese Art von Änderung sich auf eine bestehende Restbussimulation auswirken kann:

- Das betroffene Attribut wird ausschließlich intern (in R_{gen}) verwendet und hat keine Auswirkung hinsichtlich R_{user}
  $\rightarrow$ Keine Konsolidierung notwendig (R_{gen} kann einfach neu generiert werden).
- API-Namen sind betroffen. ECUs, Botschaften oder Signale weisen in K_{new} einen anderen Bezeichner auf als in K_{old}, sind aber (auf Grund einer bestimmten Äquivalenz-Metrik) die selben Elemente
  $\rightarrow$ Benutzerdefinierter Code, der sich auf API-Namen bezieht, die von den Änderungen am Bezeichner eines Elementes betroffen sind, muss angepasst werden, indem alle C-Bezeichner mit dem alten Namen durch die C-Bezeichner des neuen Namens ersetzt werden.
- API-Semantik ist betroffen. Hier muss der Benutzer benachrichtigt werden, um eventuelle Inkonsistenzen beheben zu können, die auftreten, wenn der benutzerdefinierte Code zwar syntaktisch korrekt ist, aber durch die Änderungen eine falsche Semantik erhält. Ein Beispiel für den letzteren Fall wäre die Änderung der Länge eines Signals von 2 Bit auf 4 Bit. Der für den Benutzer verwendbare Typ bleibt hierbei 1 Byte groß (C-Typ `unsigned char`), so dass alle Verwendungen des

Signals im benutzerdefinierten Code syntaktisch korrekt sind und nicht vom Compiler beanstandet werden können
→ Jedes Vorkommen der betroffenen API-Funktionen muss für den Benutzer kenntlich gemacht werden, damit dieser überprüfen kann, ob die Semantik der Verwendung der API-Funktion an der entsprechenden Stelle durch die Änderung beeinflusst wird. [18]

Diese Klassifizierung bietet keine scharfe Abgrenzung, da es Fälle gibt, bei denen der Vergleich von K_{old} und K_{new} nicht ausreicht, um zu entscheiden, ob eine bestimmte Änderung semantisch als das Hinzufügen eines neuen Elementes und das Entfernen eines alten Elementes gewertet werden soll oder als komplexe Änderung eines bestehenden Elementes. Die Intention der Änderung ist nicht aus dem statusbasierten Vergleich des alten und neuen Zustands ermittelbar. Ein operationsbasierter Vergleich, beschrieben z.B. in [Lan09], kann diesen Missstand beheben, ist jedoch nur praktikabel, wenn die Änderungen in kontrollierter und protokollierter Form vorliegen, was im vorliegenden Anwendungsfall, bei dem die Eingabedateien durch ein beliebiges Tool extern geändert werden können, nicht gegeben ist und auch nicht implementiert werden kann. Eine eindeutige Identifikation (an Hand des Namens oder eines anderen eindeutigen Attributes) von Elementen könnte diese Unschärfe der Klassifikation beheben, ist aber nicht gegeben, da für den hier vorliegenden Anwendungsfall gerade diese eindeutigen Attribute (die sich häufig in den API-Namen manifestieren) auch als variabel betrachtet werden müssen, um eine Anpassung von API-Namen im bestehenden Code durchführen zu können (so ist z.B. die Botschafts-ID für jeden Kanal eindeutig, zwischen Versionen kann sich diese jedoch ändern, so dass dieses Attribut trotz seiner Eindeutigkeit nicht zur Nachverfolgbarkeit des betroffenen Elementes verwendet werden kann). [19]

[18] Die Verwendung von Bitfeldern beseitigt das Problem nicht (Zuweisungen semantisch ungültiger Werte können dadurch weiterhin nicht vom Compiler aufgedeckt werden). Ebenso kann eine signalspezifische Typ-Definition dies nicht aufdecken, da die Typ-Definition auf einen bestehenden numerischen Wert-Typ von C abgebildet werden muss (und damit wieder keine Wertebereichs-Überprüfung durch den Compiler erfolgen kann). Eine echte Subtyp-Definition für numerische Typen mit Einschränkung des Wertebereichs, wie es beispielsweise in Ada möglich ist, existiert in C nicht. Selbstverständlich sind auch in Ada nur triviale Bereichsüberschreitungen zur Übersetzungs-Zeit erkennbar, wie etwa bei der Zuweisung von Literalen oder konstanten Ausdrücken; die Überprüfung von komplexen, dynamischen Ausdrücken muss weiterhin zur Laufzeit erfolgen.

[19] Die Generierung von künstlichen eindeutigen IDs ist prinzipiell möglich. Die Nachteile, ausführlich diskutiert in zahlreichen Arbeiten zu dieser Thematik im Kontext von Datenbanken, sind jedoch gravierend. Im Wesentlichen verschiebt sich das Problem dann auf das Problem der Aufrechterhaltung dieser IDs; die zu Grunde liegende Problematik bleibt dieselbe.

Es ist also notwendig, eine heuristische Klassifizierung von Änderungen an Hand der strukturellen Eigenschaften von K_{old} und K_{new} vorzunehmen.

Es müssen jedoch nicht nur Änderungen an den Datenbasen in die existierende Restbussimulation übernommen werden können, sondern bereits vorgenommene Anpassungen (Aktivierungsstatus von Netzwerk-Knoten und Botschaften, benutzerdefinierte Callbacks) müssen auf das neu erzeugte Modell angewandt werden, um zu vermeiden, dass der Benutzer Elemente nach dem Nachziehen von Änderungen erneut annotieren muss (z.B. Festlegen des Aktivierungsstatus von Elementen und definierte Callbacks für Elemente).

Abgesehen von dieser Übernahme vorgenommener Änderungen erfolgt die Unterstützung für Modelländerungen nur in Downstream-Richtung, d.h. es werden nur Änderungen von den Datenquellen zum Modell verfolgt; in die andere Richtung werden keine Änderungen propagiert (dies wäre auch nicht sinnvoll, da die einzigen zulässigen Ergänzungen am Modell Aspekte betreffen, die im Kontext der Datenquellen keine Bedeutung haben, z.B. Aktivierungsstatus und definierte Callbacks). Ebenfalls wird keine gesonderte Unterstützung für typische Mehrbenutzer-Konflikte implementiert (unabhängige Änderungen desselben Modells durch mehrere Benutzer mit späterer Zusammenführung der Ergebnisse).

Der grundsätzliche interne Ablauf beim Neugenerieren einer Restbussimulation in Reaktion aus Änderungen an den Eingangsdaten ist in Abbildung 3.7 dargestellt. Die einzelnen Schritte im Detail:

Eingangsdaten in Zwischenformat transformieren Die Kommunikationsmatritzen, auf die sich die Projekt-Konfiguration bezieht, werden in ein Zwischenformat transformiert.

Nachziehen von Benutzermanipulationen am alten Modell Die vom Benutzer vorgenommenen Änderungen an einem alten Zustand der Datenbasis werden, sofern vorhanden, in die neue Datenbasis übernommen. Dies betrifft bisher nur den Aktivierungs-Status von Knoten und Botschaften, ist aber prinzipiell erweiterbar auf andere Objekt-Eigenschaften.

Code-Generierung Generierung des C-Codes für R_{gen} und für neue Artefakte in R_{user}.

Differenzen-Bildung Ermitteln der Unterschiede zwischen der neuen Datenbasis und der alten Datenbasis.

Code-Konsolidierung Ausgehend von den ermittelten Unterschieden zwischen den Ständen der Datenbasis wird R_{user} (sofern existent) angepasst. Das genaue Vorgehen hierzu ist in Abschnitt 4.2.3 beschrieben.

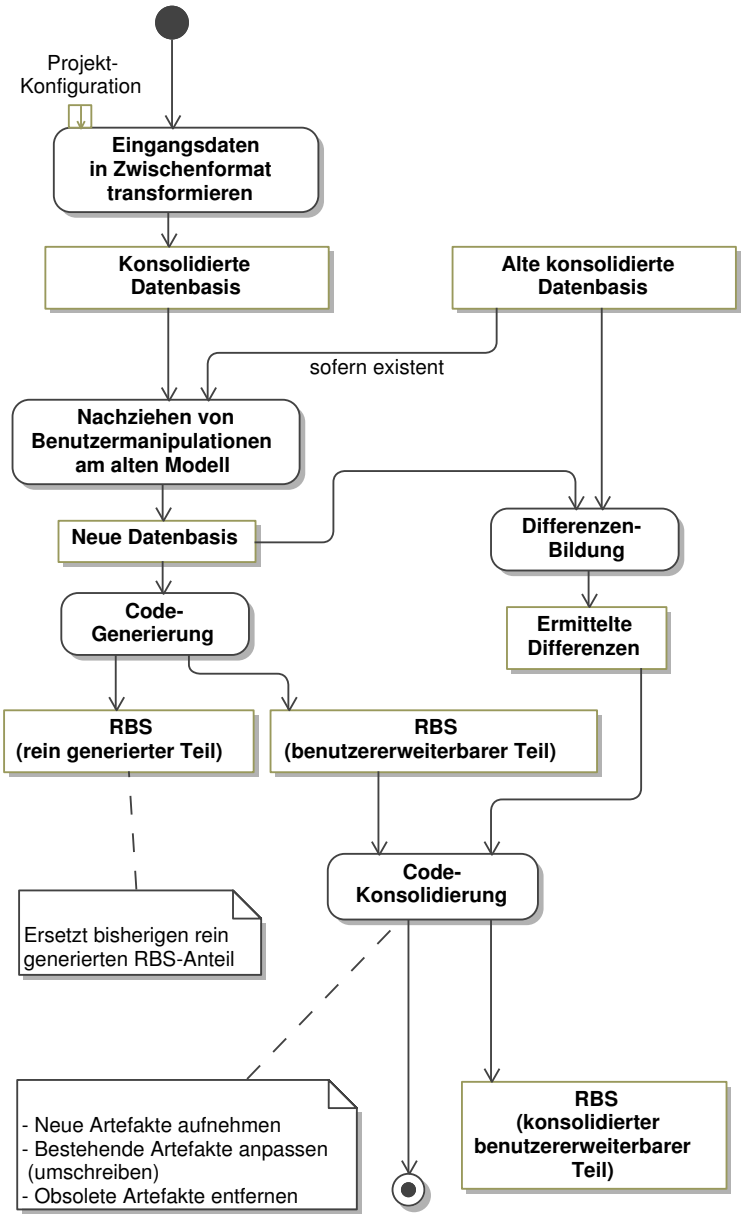

Abbildung 3.7: Reaktion auf Änderungen an Eingangsdaten [Ber]

3.5 Modularitäts-Konzept

Um die Anbindung externer Funktionalität zu realisieren, d.h. die Möglichkeit zu bieten, zusätzliche Bibliotheken einzubinden, die das Kern-System in bestimmter Weise erweitern, stehen im Java-Umfeld einige etablierte Mechanismen zur Verfügung, die im Folgenden kurz beschrieben werden; es läßt sich nicht vermeiden, dass dabei auf einige Aspekte der Classloader-Infrastruktur von Java eingegangen wird.

Allen Mechanismen gemeinsam ist, dass sie das Problem lösen, wie das Kern-System das Vorhandensein bestimmter Erweiterungen in externen Modulen registriert, wobei diese Erweiterungen immer konkrete Implementierungs-Klassen eines bestimmten Interfaces sind, dass das Kern-System zu diesem Zwecke vorsieht. Im einfachsten Fall reduziert sich dies auf die Aufgabe, alle konkreten Implementierungen eines bestimmten Interfaces zu finden, die vom Programm-Kontext aus zugänglich sind; dies ist in der Regel jedoch nicht erwünscht (z.B. wenn bestimmte Basis-Klassen nicht verwendet werden sollen, da sie keine vollständige Implementierung bereitstellen).

Die Problematik ist in Abbildung 3.8 beispielhaft dargestellt. Hier soll das Kern-System durch die Bereitstellung von Implementierungen des Interfaces `core.ISomeService` durch externe Module erweitert werden können. Das Paket `core` darf keine Abhängigkeiten von diesen zusätzlichen Modulen haben (zum Zeitpunkt der Auslieferung des Systems ist u.U. nicht einmal bekannt, dass es diese Module gibt). Ziel ist es, dass die Methode `SomeServiceLookup.getImplementations` alle Implementierungen auflisten kann, ohne direkte Kenntnis über die installierten Erweiterungs-Module zu haben.

Klassen werden in Java immer über einen so genannten *Classloader* geladen; dies erfolgt unmittelbar bevor sie das erste Mal verwendet werden.[20] Dabei wird in der Regel eine Bytecode-Datei (so genanntes *Classfile*) aus einer persistenten Form in den Speicher geladen und dynamisch mit der aufrufenden Klasse verlinkt. Um die zu ladende Klasse zu finden, wird deren voll qualifizierter Name verwendet. Der (übliche) Classloader hat einen Suchpfad (der so genannte *Classpath*), der die möglichen Speicherorte angibt (in Form von Dateisystem-Verzeichnissen oder JAR-Archiven), in denen die zu suchende Klasse gesucht wird. Es wird die erste Klasse geladen, die gefunden wird. Dies ist effizient möglich, wenn der Name der Klasse bekannt ist. Bei der naiven Suche nach allen Implementierungen eines bestimmten Interfaces müssten alle Klassen im Classpath

[20] Eine semantisch präzisere Auskunft zu den Umständen, die das Laden einer Klasse auslösen, sei auf die JVM-Spezifikation verwiesen (z.B. in [LY99]).

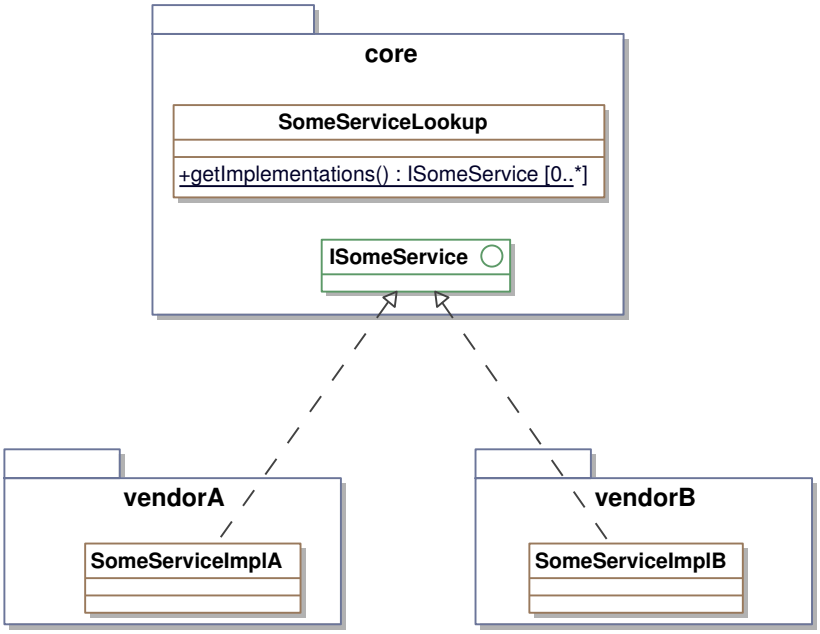

Abbildung 3.8: Generische Problematik der nicht-invasiven Erweiterung durch externe Module [Ber]

geladen werden und untersucht werden, ob sie dieses Interface implementieren; dies ist bei größeren Systemen nicht mehr effizient möglich,[21] da diese mehrere zehntausende von Classfiles enthalten können und die Suche somit unzumutbar lange dauern würde.

Aus diesem Grund ist dieser naive Algorithmus in der Praxis für größere Systeme nicht nutzbar; stattdessen muss ein deklarativer Ansatz gewählt werden, bei dem Anbieter einer Implementierung für ein bestimmtes Interface dies explizit durch Meta-Informationen bekanntmachen (und so für das Kern-System programmatisch zugänglich machen). Dies kann im Prinzip in beliebiger Weise

[21] Je nach verwendetem Framework ist eine derartige Inspektion aller Klassen überhaupt nicht möglich, z.B. in OSGi-Containern, da dort kein globaler Konsens über die ladbaren Klassen besteht, sondern jedes Modul seinen eigenen Classpath (bzw. ein Konzept äquivalent zum Classpath) verwaltet.

geschehen, z.B. auch durch einen anwendungsspezifischen eigenen Mechanismus. Typisches Vorgehen bei der manuellen Registrierung wäre (bezogen auf das dargestellte Beispiel) die Einrichtung einer Methode `registerSomeService` für die im `SomeServiceLookup`-Klasse, wobei garantiert sein muss, dass jedes Modul, das Implementierungen bereitstellt, diese Registrierungs-Methode für jede von ihm bereitgestellte Implementierung einmal aufruft, wobei weiterhin garantiert sein muss, dass alle diese Aufrufe erfolgen, bevor die Liste der Implementierungen das erste Mal abgefragt wird, was bei größeren Systemen auf Grund transitiver Abhängigkeitsbeziehungen und unterschiedlicher Aktivierungs-Zeitpunkte für einzelne Module nicht einfach zu realisieren ist, ohne explizit in die Aktivierungs-Reihenfolge der Module einzugreifen; ein solcher Eingriff hat jedoch das Potential, andere System-Eigenschaften negativ zu beeinflussen.

Um die Notwendigkeit der manuellen Registrierung zu umgehen gibt es die folgenden drei Mechanismen, die sich als Standards etabliert haben, und die jeweils eigene Einschränkungen und Möglichkeiten aufweisen:

Service Provider Interface Die mit Java 6 eingeführte `java.util.ServiceLoader`-Klasse ermöglicht das Ermitteln von Implementierungen eines Interfaces über den Classpath, wobei die Deklaration der Implementierungen wie folgt erfolgen muss: im Verzeichnis `services` im Verzeichnis `META-INF` des Archivs, das die zu registrierende Implementierung enthält, muss eine Datei angelegt werden mit dem vollständig qualifizierten Namen des betroffenen Interfaces als Dateinamen. Der Inhalt dieser Datei besteht aus einer Zeile mit dem vollständig qualifizierten Namen der Implementierungs-Klasse für jede bereitgestellte Implementierungs-Klasse, wobei sich diese nicht zwangsläufig in den Paketen befinden müssen, die von dem registrierenden Archiv verwaltet werden. Diese Art der Registrierung funktioniert prinzipiell mit allen Anwendungen, die Java 6 verwenden. Die Implementierungen für ein bestimmtes Interface können dann (bezogen auf das Beispiel) mittels des Aufrufs `ServiceLoader.load(ISomeService.class)` ermittelt werden.

OSGi-Services Da die Eclipse-Umgebung, in deren Kontext das System läuft, ein vollständiger OSGi-Container ist (Equinox), können die OSGi-Mechanismen zur Registrierung für Services für ein bestimmtes Interface genutzt werden. Siehe hierzu die OSGi-Spezifikation [OSG]. Diese Art der Registrierung funktioniert mit allen üblichen OSGi-Containern und hat prinzipiell den Vorteil, dass Services zur Laufzeit registriert und deregistriert werden können, was für den hier betrachteten Anwendungsfall jedoch irrelevant

ist [MVA10].

Eclipse Extension Registry Ein noch spezifischerer Mechanismus ist die Eclipse Extension Registry, die die Grundlage des Plugin-Konzeptes von Eclipse bildet und entsprechend nicht leicht außerhalb von Eclipse-Anwendungen nutzbar ist. Grundlage sind die so genannten Extension Points, die die erweiterbaren Aspekte des Systems deklarieren. Ein externes Modul (d.h. ein OSGi-Bundle), das einen bestimmten Extension Point verwenden will, muss in einer Datei `plugin.xml` im Bundle des Moduls ein XML-Fragment mit den von dem verwendeten Extension Point erwarteten Informationen (u.U. genügt hier ein Klassenname der von diesem Modul bereitzustellenden Implementierungs-Klasse). Die Implementierungen für jeden Extension Point sind über ein spezielles, Eclipse-spezifisches API abrufbar, das weit weniger komfortabel ist als der einfache `ServiceLoader`-Mechanismus, jedoch prinzipiell die Möglichkeit bietet, beliebige zusätzliche Strukturen und Informationen, die über den Klassennamen hinausgehen, deklarativ anzugeben. Die Notwendigkeit, dass die Existenz des Extension Point explizit angegeben werden muss sorgt dafür, dass dieser rudimentär dokumentiert ist und von potentiellen Verwendern erkannt werden kann.

Mit jedem dieser Mechanismen kann das gewünschte Ziel erreicht werden. Die OSGi-Services sind jedoch schwer korrekt zu verwenden. Die Eclipse Extension Registry macht die Anwendung abhängig von Teilen des Eclipse-Frameworks und die Verwendung ist eher umständlich. Die Verwendung von Extension Points ist jedoch zwingend notwendig für Aspekte, die vom Eclipse-Framework selbst zur Erweiterung vorgesehen sind, d.h. die Integration der Benutzerschnittstelle und der Projekt-Strukturen, der Compiler-Toolchain, so dass hier dieser Mechanismus verwendet werden muss. Als einfachste Lösung wird jedoch, wo möglich, der `ServiceLoader`-Mechanismus verwendet.

Kapitel 4

Ausgewählte Aspekte der Implementierung

4.1 IDE und Werkzeugkette

Das vorgestellte Generator-Framework wurde in eine Eclipse-basierte Entwicklungsumgebung integriert, wobei die Notwendigkeit, eine vollständige C-IDE für die benutzererweiterbaren Teile der Restbussimulation bereitzustellen, den Ausschlag für die Wahl von Eclipse als Basis-Plattform gab; im Folgenden einige der Integrationspunkte:

C-IDE Das CDT-Projekt (*C Development Tooling*) ergänzt die Eclipse-Basis-Plattform (die so genannte *Workbench*) um eine C- und C++-Entwicklungsumgebung mit Editor, Syntax-Highlighting, semantischer Navigation und Symbolauflösung, Auto-Vervollständigung sowie Compiler- und Build-Unterstützung. CDT bildet die Basis zahlreicher kommerzieller C- und C++-Entwicklungsumgebungen, darunter CodeWarrior (ab Version 10) und die QNX Momentics Suite, so dass sich für erfahrene Entwickler der Einarbeitungsaufwand in Grenzen hält.

Toolchain-Anbindung Der verwendete Cross-Compiler (eine GCC-Variante) bzw. der lokale `mingw`-Compiler wird über deklarative Schnittstellen von CDT eingebunden.

Makefile-Generierung durch CDT Makefiles für ein Projekt werden von CDT generiert, wobei die für das Projekt definierten Quellcode- und Header-Verzeichnisse als Quelle für Abhängigkeitsinformationen dienen.

Ausführung des Build durch CDT Die Makefiles für das Projekt werden von einem regulären `make`-Kommando ausgeführt und die Ausführung von CDT überwacht, wobei eventuelle Fehler in entsprechende UI-Elemente und Quellcode-Annotationen umgewandelt werden.

Deployment durch RSE Die Übertragung des RBS-Binary auf die CANyon-Zielhardware erfolgt mittels SSH, wobei eine reine Java-Bibliothek für das SSH-Protokoll zum Einsatz kommt, die Teil des RSE-Projektes (*Remote Systems Explorer*) ist.

Debugging durch CDT und RSE Das Debugging auf der CANyon-Zielhardware erfolgt mittels `gdb` und `gdbserver`, wobei `gdb` auf dem IDE-Host läuft und `gdbserver` auf dem CANyon. CDT macht die von `gdb` zur Verfügung gestellte Funktionalität in der UI der IDE zugänglich.

RBS-UI Die Informationen der konsolidierten Datenbasis werden in einer speziellen Perspektive angezeigt, so dass der Anwender einfachen Zugriff auf die RBS-API-Funktionen hat, die auf diesen Informationen möglich sind. Dies sind im Wesentlichen die Funktionen zur Signalmanipulation und zur dynamischen Aktivierung und Deaktivierung von Botschaften und Knoten. Ebenso wurden UI-Elemente integriert, die eine Navigation der benutzererweiterbaren Teile des RBS-Quellcodes mit Hilfe der semantischen Struktur der zu Grunde liegenden Datenbasis erlauben.

Kritischster Integrationspunkt war die Auflösung von Diskrepanzen zwischen der vom Eclipse-Framework vorgegebenen Projekt-Struktur und der tatsächlich benötigten Projekt-Struktur (wie in Abschnitt 3.3 beschrieben). Ein ursprünglicher Ansatz, bei dem viele der dort beschriebenen Elemente in impliziter Weise beschrieben wurden, hatte zur Auswirkung, dass eine nachträgliche Umkonfigurierung bestimmter Aspekte eines bestehenden Projektes nicht möglich waren, und zog eine große Umstrukturierung nach sich, die als Resultat das in Abschnitt 3.3 beschriebene Modell hervorbrachte, dass sich in der folgenden Entwicklung dann jedoch sehr gut bewährte und ein nahezu ideales Abstraktionsniveau aufweist.

4.2 Zwischenformat und Code-Konsolidierung

In diesem Abschnitt wird die Implementierung der Zwischendarstellung kurz vorgestellt und deren Verwendung im Hinblick auf die Anpassung des benutzererweiterten Quellcodes beschrieben.

4.2.1 EMF

Als Format für die Zwischendarstellung der konsolidierten und linearisierten Eingangsdaten wurde EMF verwendet. EMF ist ein Framework zur modellbasierten Softwareentwicklung, das als Schnittstelle zwischen Java, UML und XSD geeignet ist und das für bestimmte, häufig benötigte Aspekte (wie etwa Serialisierung und Deserialisierung in XML, XMI und ein EMF-spezifisches Binärformat, Benachrichtigung von Listenern bei Änderungen) direkt nutzbaren Java-Code generiert. Die Entscheidung auf EMF fiel aus zwei Gründen:

- Es musste ein serialisierbares Zwischenformat gefunden werden, wobei die Serialisierung und Deserialisierung im Hinblick auf die Erweiterbarkeit durch Dritt-Anbieter (u.U. mit anderen Sprachen als Java) erfolgen sollte, so dass sich ein XML-basiertes Format anbot. Zu den Alternativen zu EMF in dieser Hinsicht zählen z.B. die zahlreichen Implementierungen der JAXB-Spezifikation.

- Das Zwischenformat sollte die in Abschnitt 3.4.5 umrissenen Konzepte zur Ermittlung und Behandlung von Änderungen ermöglichen. Für EMF existiert hier das im nächsten Abschnitt beschriebene *EMF Compare*. Hierzu gibt es sehr wenige Alternativen; die Bibliotheken `XMLDiff` und `XMLUnit` ermöglichen das Bilden von Differenzen zwischen zwei XML-Dateien (desselben Schemas), waren aber von der Granularität her nicht geeignet, um im Kontext dieser Arbeit verwendet werden zu können.

4.2.2 EMF Compare

Zur Anpassung bestehenden benutzererweiterbaren Quellcodes in Reaktion auf eine erfolgte Änderung an den Eingangs-Kommunikationsmatrizen des Projektes werden die Stände der konsolidierten Datenbasis vor dem letzten Generieren und der aktuelle Stand verglichen (siehe Abbildung 3.7). Dies erfolgt mittels *EMF Compare*, das ein generisches API zum Vergleich von EMF-Modellen bereitstellt. Die grundsätzliche Vergleichs-Heuristik von Modell-Elementen, die von EMF Compare realisiert wird, ist in [XS05] ausführlich und in [Lan09] kurz beschrieben; einbezogen in die Vergleichs-Operationen werden unter anderem Objekt-Relationen und Objekt-Eigenschaften. Die Ermittlung der Differenzen erfolgt in zwei Schritten: die Bildung des so genannten `MatchModel`, das korrespondierende Elemente zwischen zwei Modell-Ständen (und Elemente ohne korrespondierendes Element im jeweils anderen Stand) liefert, und die Bildung

des `DiffModel`, das das `MatchModel` transformiert und um detailliertere Informationen zu den Unterschieden zwischen korrespondierenden Elementen erweitert.

Die Rechenzeit für die Gewinnung der `MatchModel`- und `DiffModel`-Instanzen zwischen zwei Versionen der konsolidierten Datenbasis ist in nicht-linearer Weise abhängig von der Größe der Datenbasis. EMF Compare bietet die Möglichkeit, eine obere Grenze für die Komplexität der Ähnlichkeits-Berechnung festzulegen, mit dem Risiko, dass manche Änderungen nicht zugeordnet werden können.[1] Für die konkret untersuchten realen Datenbasen war die Performance des Standard-Algorithmus durchaus akzeptabel, so dass dieser beibehalten wurde. In [FL95] ist jedoch ein alternativer Algorithmus beschrieben, der in einem Projekt zur Ergänzung von EMF Compare verwendet wird [Leo].

4.2.3 Gewinnung von Konsolidierungs-Informationen

Für die Gewinnung der Konsolidierungs-Informationen wird das `DiffModel` zwischen altem und neuem Modell erfasst, die so gewonnenen Änderungs-Informationen werden klassifiziert (in nicht relevante, konsolidierbare und nicht konsolidierbare Änderungen) und die betroffenen Elemente werden expandiert, indem zu jedem betroffenen Element die zugehörigen API-Namen, wie sie im alten Modell auftreten und die zugehörigen API-Namen, wie sie im neuen Modell existieren, aufeinander abgebildet (äquivalent für die Artefakt-Namen); die Abbildung ist eine einfache String-zu-String-Abbildung von altem API- oder Artefakt-Name zu neuem API- oder Artefakt-Name für jeden Aspekt des von der Änderung betroffenen Elementes. Die so gewonnenen Mapping-Informationen werden verwendet, um die existierenden Quellcode-Artefakte in R_{user} anzupassen, wie in den folgenden Abschnitten beschrieben. In der momentanen Implementierung erfolgt die Ermittlung der API- und Artefakt-Namen über bestimmte Template-Fragmente, die mit alter und neuer Element-Version instantiiert werden und in die benötigten Mapping-Informationen umgewandelt werden. Die Integration in die Templates ermöglicht das Single-Sourcing dieser Informationen, da diese ohnehin bereits in den Templates definiert sind.

4.2.4 Konsolidierung von API-Namen

Für jedes Artefakt, das Teil von R_{user} ist, werden die gewonnenen Mapping-Informationen angewandt, indem alle C-Bezeichner, die einem in den Mapping-

[1] Dies erfolgt durch die Definition der Größe des Suchfensters, innerhalb dessen nach einem potentiell ähnlichen Element gesucht wird. Die Distanzmetrik ist in ebenfalls in der aufgeführten Arbeit beschrieben.

Informationen enthaltenen Schlüssel entsprechen (entsprechend einem alten API-Namen), in die diesem Schlüssel entsprechenden Wert (entsprechend dem neuen API-Namen) umgeschrieben werden. Hierbei wird bisher rein lexikalisch vorgegangen, d.h. dort, wo ein Lexem auftritt, dass dem zu ersetzenden Element entspricht, wird dieses ersetzt; eine Ersetzung, die auf der abstrakten Syntax von C operiert und so z.B. keine Ersetzungen in auskommentiertem Code vornimmt, wäre jedoch möglich und wünschenswert. Die Beachtung der lexikalischen Struktur bewirkt gegenüber der naiven Ersetzung, dass keine Teilersetzungen vorgenommen werden, sondern nur die Stellen behandelt werden, an denen der zu ersetzende Bezeichner tatsächlich auftritt und nicht nur als Teil eines anderen Bezeichners; das Fehlen von Namensräumen in C und die damit verbundene notwendige volle Qualifizierung von Bezeichnern (z.B. durch Unterstriche im Bezeichner) erleichtert diesen Vorgang.

Darüber hinaus gibt es den Fall von gegenseitiger Ersetzung, d.h. ein Element A wird in B umbenannt und Element B wird im gleichen Transformations-Schritt in A umbenannt. Hier würde die beschriebene Vorgehensweise scheitern, da die Ersetzung von A zu B, gefolgt von einer nachfolgenden Ersetzung von B zu A zur Folge hätte, dass alle Vorkommen von A und B zu A ersetzt werden. Um dieses Problem zu lösen, muss die Ersetzung in zwei Stufen erfolgen: die Ersetzung des zu ersetzenden Elementes durch einen eindeutigen temporären Bezeichner und nachfolgend eine Ersetzung dieses temporären Bezeichners durch den endgültigen Bezeichner.

4.2.5 Konsolidierung von Artefakt-Namen

Für das Umbenennen von bestehenden Quellcode-Dateien werden die Mapping-Informationen ebenfalls genutzt. Bei der Behandlung von Änderungen, die, wie oben beschrieben, eine gegenseitige Umbenennung von Artefakten bewirken, wird eine Variante des zuvor beschriebenen zweistufigen Prozesses angewandt (auf der Ebene der Artefakt-Namen). Die in Abschnitt 3.4.2 beschriebenen Mechanismen können angewandt werden, um, wie dort beschrieben, die existierenden Dateien zu sichern und das Überschreiben von existierenden Dateien bei gleichem Inhalt zu verhindern.

4.2.6 Benutzerinteraktion

In der vorliegenden Implementierung werden nicht konsolidierbare Änderungen, sofern der Generator im Kontext einer IDE läuft, dem Benutzer mittels Markierungen (*Resource Marker* in der Eclipse-Terminologie) kenntlich gemacht durch

graphische Annotationen der Stellen, an denen sich die von der Änderung betroffenen API-Elemente befinden. Beim Wegfall eines Signals aus der Datenbasis wären dies etwa alle Stellen in R_{user}, an denen Aufrufe der (nun nicht mehr vorhandenen) Signal-Zugriffsmethoden erfolgen. Die dem Benutzer präsentierten Informationen sind generischer Natur und warnen, dass Änderungen des API erfolgten und diese vom Benutzer überprüft werden müssen. Hier wäre eine bessere Klassifizierung der Änderungen oder eine genauere Fehlerbeschreibung (insbesondere bei den erwähnten Änderungen an der Signal-Kodierung) vorteilhaft.

4.3 Dynamische Aspekte

Um eine generierte Restbussimulation sinnvoll ausbauen zu können und nicht auf externe Tools zur Überwachung der durch die Restbussimulation erzeugten Buskommunikation angewiesen zu sein, wurden verschiedene Mechanismen implementiert und in die Entwicklungsumgebung integriert.

4.3.1 CAN-Analyzer

Es wurde ein TCP/IP-basierter Kommunikationskanal zur Weiterleitung der CAN-Kommunikation der auf einem CANyon-Gerät angeschlossenen Busse realisiert; dadurch lassen sich Botschaften und Signale überwachen, ohne dass am Entwickler-PC eine extra CAN-Hardware vorhanden ist (Tunnelung von CAN nach TCP/IP via CANyon).

4.3.2 Dynamische Beeinflussung

Ebenfalls über TCP/IP wurde ein Kommunikationskanal zur dynamischen Beeinflussung einer laufenden Restbussimulation realisiert, über den die innerhalb der Simulation laufenden Netzwerkknoten aktiviert und deaktiviert werden können, das Sendeverhalten einzelner Botschaften verändert werden kann und gesendete Signale neu belegt werden können.

Kapitel 5

Zusammenfassung und Ausblick

Die realisierten Generatoren für die unterstützten Zielplattformen ermöglichen eine einfache Generierung von Restbussimulationen und die Erweiterung dieser Restbussimulationen auf eine klar definierte Weise. Indem die Variabilität der Datenbasen, die als Grundlage für die generierte Restbussimulation dienen, im Generierungs-Prozess durchgängig berücksichtigt wird, wird die Anpassung von generierten und erweiterten Restbussimulationen in Reaktion auf Änderungen dieser Eingangsdaten erleichtert.

Das entwickelte Generator-Framework eignet sich als Grundlage für die Implementierung anwendungsspezifischer oder organisationsspezifischer Generatoren (entweder als neue Plattformen oder durch Spezialisierung der Generatoren für bestehende Plattformen) mit vertretbarem Aufwand und, bei Beachtung gewisser Rahmenbedingungen, ohne Änderungen am Kern-System.

Im Folgenden werden einige mögliche und empfehlenswerte Erweiterungen für das entwickelte Generator-Framework beschrieben, die die in dieser Arbeit begonnenen Aspekte verbessern oder ergänzen.

5.1 Zusätzliche Ziel-Plattformen

Die bisher vollständig (CANyon) oder teilweise (durch die Vector XL Library unterstützte Plattformen) unterstützten Ziel-Systeme sind nur in Verbindung mit einem Betriebssystem verwendbar. Wie in Abschnitt 2.4 beschrieben, werden diese unterstützten Ziel-Plattformen durch mächtigere APIs angesprochen, als sie

für Systeme ohne Betriebssystem zu erwarten oder möglich sind. Eine Portierung auf eine bare-metal Hardware-Plattform unter Verwendung microcontrollernaher Programmierung würde u.U. Fälle aufdecken, für die die entworfene und implementierte Generator-Architektur noch nicht optimal ist. Eine mögliche Ziel-Plattform wäre das ISIM flexible von Berger Elektronik.

Eine weitere lohnende Zielplattform wäre eine AUTOSAR-Architektur, d.h. die Verwendung eines (extern generierten) AUTOSAR-COM-Stacks als Kommunikations-Schnittstelle, wobei die in Abschnitt 2.6.8 beschriebenen Einschränkungen zu beachten sind. Eine vollständige Generierung eines AUTOSAR-COM-Stacks liegt nicht mehr in der Domäne von Restbussimulationen.

5.2 Modellierung mittels Zustandsmodellen

Die Anwendungsfälle, für die Restbussimulationen eingesetzt werden, sind, wie bereits beschrieben, stark ereignisgesteuert und stark zustandsorientiert. Derartige Systeme können durch Modellierungs-Notationen für Zustandsautomaten gut und relativ vollständig beschrieben werden, wobei sich die graphischen Varianten dieser Notationen auch von Nicht-Programmierern nutzen lassen und so der potentielle Anwenderkreis erweitert werden kann. Durch die auf diesen Notationen möglichen Analysen lassen sich bestimmte Eigenschaften des Systems einfach automatisch untersuchen und verifizieren, die sonst durch manuelle Tests überprüft werden müssen. Ein triviales Beispiel für derartige Eigenschaften, die auf diese Weise analysiert werden können, sind unerreichbare Zustände.

Etablierte Notationen sind hier UML Statecharts [Obj] (die z.B. durch das Papyrus-Projekt [Eclf] in die entwickelte Werkzeugkette integriert werden könnten) oder (als kommerzielles Produkt) die Stateflow-Erweiterung für Matlab Simulink [Mat], mittels der extern Code für das Zustandsmodell generiert werden kann und dieser über eine zu definierende, schmale Schnittstelle an die generierte Restbussimulation angebunden werden könnte.

Das in [Sam08] beschriebene Event-Framework wäre ebenfalls ein mögliches Ziel für die Integration der Restbussimulation in einen Zustandsautomaten.

Die wichtigsten Aspekte, die im Zusammenhang mit der Integration von Zustandsautomaten in die Restbussimulation behandelt werden müssen, sind im Folgenden dargelegt.

5.2.1 Nutzung des RBS-API

Das API der Restbussimulation (Signalmanipulation, Knoten- und Botschafts-Aktivierung) muss vom Zustandsautomaten aus aufrufbar sein, damit Guards sich auf Signalwerte beziehen können und innerhalb von Aktionen die entsprechenden Methoden zum Aktivieren und Deaktivieren von Elementen oder zum Setzen neuer Signalwerte verwendet werden können. Ausreichend ist hier die Einbindung des generierten Restbussimulations-API durch den Zustandsautomaten; nicht zwingend notwendig, aber aus Sicht der Benutzerfreundlichkeit wichtig, ist die Repräsentation der Modell-Elemente der Restbussimulation im Modellierung-Werkzeug, so dass nicht erst beim Übersetzen oder Linken der RBS gegen den Zustandsautomaten die Verwendung von falschen Bezeichnern oder Funktionsnamen aufgedeckt wird, sondern der Benutzer bereits beim Modellieren die gültigen RBS-API-Namen zur Verfügung hat. Im Falle von Werkzeugen, die auf UML Statecharts operieren, wäre dies z.B. einfach möglich, indem die Funktionen der Restbussimulation durch Operationen einer Klasse `RBS` im Kontext des Statecharts symbolisch zugänglich gemacht werden. Dazu kann z.B. ein Transformations-Schritt vom internen Zwischenformat in das üblicherweise vom UML-Tools verwendeten XML-Metadata-Interchange-Format (XMI) implementiert werden; diese Informationen können dann vom entsprechenden UML-Tool importiert werden, bevor ein Statechart angelegt wird.

5.2.2 Weiterleitung von RBS-Ereignissen

Der Zustandsautomat muss durch die Restbussimulation mit einem Ereignisstrom versorgt werden. Die Abbildung von Ereignissen der Restbussimulation in die erwarteten Strukturen des Zustandsautomaten muss implementiert werden, so dass sich die Ereignisse als Events für Transitionen im Sinne der Statechart-Semantik nutzen lassen.

5.3 AST-basierte Code-Konsolidierung

Die in Abschnitt 4.2.3 beschriebene Konsolidierung arbeitet auf textueller Ebene, was teilweise fragil in Bezug auf die Struktur des Quellcodes ist. Eine intelligentere Implementierung könnte auf Syntaxbaum (AST) arbeiten, der dem Code zu Grunde liegt; so könnte z.B. kommentierter Code von der Konsolidierung ausgeschlossen werden oder selektiv einzelne Methoden einer Datei überschrieben

werden, sowie die Robustheit gegenüber Änderungen der Quellcode-Struktur erhöht werden.

Als funktionales Vorbild für die Funktionsweise einer derartigen Implementierung könnte das im Rahmen von JET verwendete JMerge [Eclb] dienen, dass die Manipulation eines Java-ASTs auf einem relative hohen Abstraktionsgrad ermöglicht. Auf Grund der höheren syntaktischen und semantischen Komplexität von C dürfte der Aufwand für die Implementierung einer funktional äquivalenten Bibliothek für C jedoch nicht unerheblich sein, selbst wenn auf bestehende Parser und AST-Bibliotheken zurückgegriffen wird (z.B. den C-Parser von CDT [Eclc]). Die enorme Komplexität ist teilweise zurückzuführen auf die lexikalisch orientierte Arbeitsweise des C-Präprozessors; hierbei gilt die Aussage von Bjarne Stroustrup (in Bezug auf den Präprozessor von C und C++, im folgenden Zitat Cpp genannt): „In retrospect, maybe the worst aspect of Cpp is that it has stifled the development of programming environments for C. The archaic and character-level operation of Cpp makes nontrivial tools for C and C++ larger, slower, less elegant, and less effective than one would have thought possible.“ [Str94] [1]

5.4 Gateway-Generierung

Mit dem entwickelten Framework sind Gateways leicht programmierbar, da alle notwendigen Kommunikations-Strukturen und Signalmanipulations-Routinen bereits generiert werden. Damit stehen alle Schnittstellen für das programmatische Routen von Dateninhalten (auf Botschafts- oder Signal-Ebene) zur Verfügung. Für einfache Gateway-Anwendungen können so manuell Mappings auf Botschaftsebene oder Signalebene eingefügt werden, indem in den Empfangs-Callbacks des Ursprungs-Kanals die notwendigen Mappings durch einfache API-Aufrufe durchgeführt werden. Sind viele Botschaften oder Signale zu routen, ist dies eine wenig produktive Tätigkeit. Eine graphische Unterstützung würde den Komfort hier deutlich erhöhen. Darüber hinaus könnte so auch eine Validierung der Konsistenz des Signal-Mappings integriert werden, die bei semantisch ungültigen Abbildungen (z.B. zwischen Signalen mit unterschiedlichen Dimensionen oder Signale mit unterschiedlichen definierten SI-Einheiten ohne Durchführung der notwendigen Skalierung) oder bei Abbildungen mit sehr großem Präzisionsverlust dabei helfen kann, bestimmte Arten von Fehlern zu vermeiden.

[1] Man beachte auch Bjarne Stroustrups Stellungnahme zu bestimmten Zitaten, die ihm fälschlicherweise zugeordnet werden (unter `http://www2.research.att.com/~bs/bs_faq.html#really-say-that`).

AUTOSAR erlaubt die Definition von Gateways als Teil einer AUTOSAR-System-Configuration (auf Botschafts-Ebene, PDU-Ebene und Signal-Ebene) [AUTa]. Dazu muss der Gateway jedoch als ECU in der System-Configuration existieren; ad-hoc Gateways (z.B. zur Integration von Messtechnik in ein existierendes Gesamt-System) lassen sich nicht definieren (dies würde auch dem Konzept von AUTOSAR als ganzheitliche Architektur für ein Gesamt-System widersprechen), so dass diese Methode sich lediglich als optionaler Konfigurations-Mechanismus für Gateways eignet, jedoch nicht alle Anwendungsfälle abdecken kann.

In [Obe07] wird ein modellbasierter Ansatz für die Konfiguration von Gateways im Automotive-Umfeld vorgestellt, der in [Obe09] durch eine Methodik zum Fault-Containment ergänzt wird; insbesondere das in diesen Arbeiten verwendete Modell zur Datenhaltung innerhalb der Gateway-ECU (in den Arbeiten Real-Time-Database genannt), das zusätzlich zu den abzubildenden Daten die Aktualisierungs-Semantik dieser Daten und andere zeitliche Aspekte verwaltet, ist auch für die allgemeine Restbussimulation interessant (wobei die temporäre Daten-Semantik bei Verwendung eines AUTOSAR-Kommunikations-Stacks dort im Wesentlichen bereits vorhanden ist, so dass hier darauf geachtet werden muss, keine parallelen Mechanismen einzuführen).

5.5 Test-Automatisierung

Die Validierung einer Umgebung zur Restbussimulation wirft gegenüber dem Test regulärer eingebetteter Systeme zusätzliche Probleme auf, da keine generische Bewertung eines kompletten Systems stattfinden kann, sondern immer nur die konkrete Implementierung auf Seiten des Benutzers gegen dessen konkrete Anforderungen getestet werden kann. Das breite Anwendungsspektrum, das durch Restbussimulationen abgedeckt werden kann, bewirkt, dass nicht der gesamte Lösungsraum für den Toolhersteller zugänglich ist, wobei dies für die funktionalen Aspekte (was macht die vom Benutzer mit dem Tool generierte und angepasste Restbussimulation?) sowie die so genannten nicht-funktionalen Aspekte (welche Bus-Last und Antwortzeiten erwartet der Benutzer von einer konkreten Restbussimulation?) der Lösung gilt. Wie bei jedem Test muss durch die Bildung von Äquivalenzklassen bestimmter Restbussimulations-Profile eine möglichst gute Abdeckung des Lösungsraumes angenähert werden. Es muss also eine Lösung gefunden werden, die Variabilität unter diesen Äquivalenzklassen in den Griff zu bekommen und die Definition von Tests für jede Äquivalenzklasse möglichst einfach zu gestalten.

Diese Methode kann bei entsprechendem Entwurf mit verhältnismäßig wenig

Aufwand auch in einer für den Benutzer geeigneten Form angeboten werden, so dass dieser eine von ihm entwickelte Restbussimulation gegen die nur ihm zugänglichen Anforderungen testen kann. Die Integration einer solchen Test-Umgebung hat den Vorteil, dass keine externen Tools (z.B. CANoe und dessen Test-Umgebung) zur Validierung der Restbussimulation verwendet werden müssen, sondern eine durchgehende Werkzeugkette angeboten werden kann, die Entwicklung, Testfall-Definition und System-Test umfasst.

5.5.1 Trace-basierter Test

Da für eine Restbussimulation als Korrektheits-Kriterium das nach außen sichtbare Verhalten gilt, kann die Validierung als Black-Box-Test erfolgen, wobei die Black-Box die generierte Software (mit gegebenenfalls vorhandenen Erweiterungen gemäß der konkreten Anwendung) und die Hardware umfasst; die Einbeziehung der Hardware ermöglicht die Verwendung desselben Testfalls für jede Hardware, die durch den Code-Generator unterstützt wird. Dadurch ergibt sich als zusätzlicher möglicher Anwendungsfall die Bewertung unterschiedlicher Hardware-Lösungen für eine konkrete Restbussimulation (z.B. für Vorab-Analysen zur Auslegung der Hardware-Lösung).

Das externe Verhalten einer Restbussimulation kann durch das Mitschneiden sämtlichen Busverkehrs zwischen Restbussimulation und anderen Komponenten vollständig erfasst werden (so genannter Trace), indem die übertragenen Nachrichten (Parameter und Dateninhalt sowie zugehöriger Zeitstempel) aufgezeichnet werden. [2] Die Validierung einer Restbussimulation kann dann durch die Auswertung von Traces in Bezug auf die für die Restbussimulation definierten Anforderungen erfolgen, d.h. für jede Anforderung kann ein entsprechendes Antwort-Verhalten der Restbussimulation in Form von erwarteten Reaktionen, die im Trace sichtbar sind, definiert werden. Die Stimulation der zu testenden Restbussimulation kann z.B. durch ein zweites System erfolgen, für das explizit zu diesem Zweck aus den Testfällen entsprechender Code generiert wird.

Allgemeine Formalismen zur Beschreibung temporalen Verhaltens existieren z.B. als Derivate von Konzepten wie CTL (*Computational Tree Logic*) oder LTL (*Linear Temporal Logic*), die beide eine (diskrete) temporale Prädikatenlogik verwenden [BK08]. Weniger theoretische Ansätze zur temporalen Logik finden sich in [HLP08].

[2] Anforderungen, die interne Zustandsänderungen der Restbussimulation betreffen, müssen in irgendeiner Form nach außen geführt werden, wenn sie mit dieser Methodik beobachtet werden sollen.

5.5.2 Graphische Test-Fall-Modellierung

Die Beschreibung von Test-Fällen mit textuellen Notationen oder Test-Skripten wird zunehmend verdrängt durch die Verwendung graphischer Notationen für die Test-Fall-Modellierung. Häufig, z.B. beim EXAM-Framework des Herstellers Micronova, wird hierfür eine Untermenge der UML verwendet (im speziellen Sequenzdiagramme), um Request-Response-Szenarien zu definieren und so das Verhalten des getesteten Systems zu prüfen.

Eine erste, sehr rudimentäre Integration von der auf dem CANyon laufenden Restbussimulation mit EXAM wurde begonnen. Die Methodik für die Integration sieht hier vor, dass eine Schnittstelle definiert wird, die die möglichen Interaktionen mit dem System beschreibt, sowie eine Implementierung dieser Schnittstelle bereitstellt. Die Implementierung muss in der Sprache Python erfolgen. Bestimmte Schnittstellen sind im so genannten EXAM-Core bereits definiert und sollten bevorzugt verwendet werden, um eine Austauschbarkeit der Tests zu gewährleisten.

Der Test-Fall-Bearbeiter kann diese Schnittstellen verwenden, um mittels UML-Sequenz-Diagrammen den Test-Ablauf zu definieren. Diese Test-Fälle können parametrisiert und ausgeführt werden; Resultate der Ausführung können in die von EXAM bereitgestellte Test-Datenbank geschrieben und so dokumentiert werden.

5.6 Clustering von CANyon-Geräten

Für umfangreiche Gesamtfahrzeug-Simulationen, insbesondere wenn die Plattform für das Rapid-Prototyping von stark algorithmenlastigen Anwendungen verwendet wird (Beispiel: Stabilitätsprogramme wie ESP), ist ein einzelnes Gerät unter Umständen zu leistungschwach, um zusätzlich zur Restbussimulation noch unoptimierte nummerische Algorithmen unter weichen Echtzeitbedingungen auszuführen. Durch eine Verteilung der Aufgaben auf mehr als ein Gerät, z.B. durch die Auslagerung von nicht unmittelbar für die algorithmische Simulation wichtigen Restbussimulation-Teilen auf ein Zweit-Gerät, kann eine Entlastung erreicht werden. Eine Verbindung der Geräte ist durch die Verwendung desselben Busses gegeben, so dass die Verwendung eines zusätzlichen Kommunikationsmediums entfällt. Für echte Hardware-in-the-Loop-Anwendungen mit Präzisions-Anforderungen im Mikrosekunden-Bereich dürfte diese Lösung eher uninteressant sein, da der Kernel des CANyon-Systems durchsatzorientiert ist und keine harten Echtzeit-Anforderungen unterstützt.

Glossar

A/D	Analog/Digital
API	Application Programming Interface
AST	Abstract Syntax Tree; strukturierte Repräsentation von Dokumenten-Inhalten
$AUTOSAR$	Automotive Open System Architecture
BCM	Broadcast Manager; Kernel-Modul von SocketCAN, das eine auftragsbasierte Schnittstelle zur Konfiguration der CAN-Kommunikation bereitstellt
BSW	Basis-Software; anwendungsunabhängige Teile eines Software-Systems
CAN	Controller Area Network; Multi-Master-Bussystem zur Kommunikation zwischen Steuergeräten
$CANyon$	Hardwareplattform von Berger Elektronik für autonome Restbussimulationen
CDT	C/C++ Development Tooling
$CSMA/CA$	Carrier Sense Multiple Access with Collision Avoidance; Buszugriffs-Verfahren, bei dem der Status des Mediums (in der Regel ist dies der Spannungspegel) verwendet wird, um Kollisionen zu erkennen und zu vermeiden
CTL	Computational Tree Logic
ECU	Electronic Control Unit; Steuergerät
EMF	Eclipse Modeling Framework

$Fibex$	Fieldbus Exchange Format
GCC	GNU Compiler Collection
GDB	GNU Debugger
HIL	Hardware-in-the-loop
I/O	Input/Output
JVM	Java Virtual Machine
LDF	LIN Description File
LIN	Local Interconnect Network; serielles Bussystem zur Kommunikation zwischen Steuergeräten und deren lokalen Aktoren und Sensoren
LTL	Linear Temporal Logic
$MOST$	Media Oriented Systems Transport; optisches Bussystem mit hoher Bandbreite, optimiert für Streaming-Anwendungen
NCF	Node Capability File
$OSGi$	OSGi Service Platform; Komponenten- und Service-Modell für Java-Plattformen
PDU	Protocol Data Unit
$POSIX$	Portable Operating System Interface for Unix
RBS	Restbussimulation
RSE	Remote Systems Explorer
RTE	Runtime Environment; Laufzeitumgebung von AUTOSAR im Kontext einer spezifischen ECU
$TDMA$	Time Division Multiple Access; Buszugriffs-Verfahren, bei dem a-priori Zeitschlitze für bestimmte Knoten oder Datenelemente reserviert werden und so ein deterministischer Buszugriff möglich ist
UI	User Interface (Benutzerschnittstelle)
VFB	Virtual Functional Bus

XMI	XML Metadata Interchange; von der Object Management Group (OMG) definiertes Austauschformat für UML-Modelle, das inzwischen auch für die Serialisierung anderer Modelle genutzt wird)
XSD	XML Schema Definition

Literaturverzeichnis

[Apa] APACHE SOFTWARE FOUNDATION: *Apache Ant*. `http://ant.apache.org`. – Stand 26.6.2011

[AUTa] AUTOSAR DEVELOPMENT COOPERATION: *AUTOSAR COM-Stack 4.0.0 (AUTOSAR_SWS_COM.pdf)*. – Stand 26.6.2011

[AUTb] AUTOSAR DEVELOPMENT COOPERATION: *AUTOSAR Specification 4.0.0*. `http://www.autosar.org`. – Stand 26.6.2011

[AUTc] AUTOSAR DEVELOPMENT COOPERATION: *AUTOSAR SWC-Specification 4.0.0 (AUTOSAR_TPS_SoftwareComponentTemplate.pdf)*. – Stand 26.6.2011

[AUTd] AUTOSAR DEVELOPMENT COOPERATION: *AUTOSAR System-Configuration 4.0.0 (AUTOSAR_TPS_SystemTemplate.pdf)*. – Stand 26.6.2011

[Ber] BERGER ELEKTRONIK GMBH: *Interne Grafik*. – Grafik für diese Arbeit erstellt

[BK08] BAIER, Christel ; KATOEN, Joost-Pieter: *Principles of Model Checking*. The MIT Press, 2008. – ISBN 9780262026499

[Ecla] ECLIPSE FOUNDATION: *ATL - A Model Transformation Technology*. `http://www.eclipse.org/atl/`. – Stand 26.6.2011

[Eclb] ECLIPSE FOUNDATION: *Beschreibung von JMerge*. `http://wiki.eclipse.org/JET_FAQ_How_does_JMerge_work%3F`. – Stand 26.6.2011

[Eclc] ECLIPSE FOUNDATION: *C Development Tooling*. `http://www.eclipse.org/cdt/`. – Stand 26.6.2011

[Ecld] ECLIPSE FOUNDATION: *Eclipse Modeling Framework Project (EMF)*. `http://www.eclipse.org/emf/`. – Stand 26.6.2011

[Ecle] ECLIPSE FOUNDATION: *Modeling Workflow Engine*. `http://www.eclipse.org/modeling/emft/?project=mwe`. – Stand 26.6.2011

[Eclf] ECLIPSE FOUNDATION: *Papyrus Projekt*. `http://www.eclipse.org/modeling/mdt/papyrus/`. – Stand 26.6.2011

[Ele] ELEKTROBIT AUSTRIA GMBH: *Dokumentation zu Tresos*. – Stand 26.6.2011

[FL95] FALOUTSOS, Christos ; LIN, King-Ip: *FastMap: A Fast Algorithm for Indexing, Data-Mining and Visualization of Traditional and Multimedia Datasets*. 1995

[Fra10] FRANK, Philipp M.: *Praktikumsbericht Berger Elektronik GmbH (intern)*. 2010

[GHJ94] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph E.: *Design Patterns. Elements of Reusable Object-Oriented Software*. 1. Auflage (Nachdruck). Addison-Wesley Longman, Amsterdam, 1994. – ISBN 9780201633610

[GN06] GLEIXNER, Thomas ; NIEHAUS, Douglas: *Hrtimers and Beyond: Transforming the Linux Time Subsystems*. 2006

[Hal11] HALLINAN, Christopher: *Embedded Linux Primer: A Practical Real-World Approach*. 2. Auflage. Prentice Hall, 2011. – ISBN 9780137017836

[Har] HARTKOPP, Oliver: *Dokumentation zu SocketCAN*. `http://www.kernel.org/doc/Documentation/networking/can.txt`. – Stand 26.6.2011

[Hit10] HITZLER, Bastian: *Konzeptionierung eines universellen Datenloggers und Simulators für Kfz-Bussysteme*. 2010. – Diplomarbeit, Fachhochschule Aachen, Fachbereich Maschinenbau und Mechatronik

[HLP08] HARMELEN, Frank van ; LIFSCHITZ, Vladimir ; PORTER, Bruce: *Handbook of Knowledge Representation (Foundations of Artificial Intelligence)*. Elsevier Science, 2008. – ISBN 9780444522115

[Hub08] HUBER, Philipp: *The Model Transformation Language Jungle - An Evaluation and Extension of Existing Approaches*. 2008. – Master Thesis, TU Wien

[jax] *JAXB Reference Implementation Project*. `http://jaxb.java.net/`. – Stand 26.6.2011

[Keg] KEGEL, Dan: *The C10K Problem*. http://www.kegel.com/c10k.html. – Stand 26.6.2011

[Ker10] KERRISK, Michael: *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. 1. Auflage. No Starch Press, 2010. – ISBN 9781593272203

[KF09] KINDEL, Olaf ; FRIEDRICH, Mario: *Softwareentwicklung mit AUTOSAR: Grundlagen, Engineering, Management in der Praxis*. 1. Auflage. dpunkt Verlag, 2009. – ISBN 9783898645638

[KW07] KINDLER, Ekkart ; WAGNER, Robert: *Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios*. 2007

[Lan09] LANGER, Philip: *Konflikterkennung in der Modellversionierung*. 2009. – Diplomarbeit, TU Wien

[Leo] LEOPOLD, Stefan: *GSoc project on improvements of model compare match engine (EMF Compare)*. http://code.google.com/a/eclipselabs.org/p/model-compare-match-engine-playground/. – Stand 26.6.2011

[LIN] LIN KONSORTIUM: *LIN Spezifikation*. http://www.lin-subbus.org. – Stand 26.6.2011

[LY99] LINDHOLM, Tim ; YELLIN, Frank: *The Java Virtual Machine Specification Second Edition*. 2. Auflage. Prentice Hall, 1999. – ISBN 9780201432947

[Mat] MATHWORKS: *Stateflow*. http://www.mathworks.com/products/stateflow/. – Stand 26.6.2011

[MVA10] MCAFFER, Jeff ; VANDERLEI, Paul ; ARCHER, Simon: *OSGi and Equinox: Creating Highly Modular Java Systems*. 1. Auflage. Addison-Wesley Professional, 2010. – ISBN 9780321585714

[Obe07] OBERMAISSER, Roman: A Model-Driven Framework for the Generation of Gateways in Distributed Real-Time Systems. In: *Proceedings of the 28th IEEE International Real-Time Systems Symposium*. Washington, DC, USA : IEEE Computer Society, 2007 (RTSS '07)

[Obe09] OBERMAISSER, Roman: *Fault and Error Containment of Gateways in Distributed Real-Time Systems*. September 2009. – TU Wien

[Obj] OBJECT MANAGEMENT GROUP: *Unified Modeling Language 2.3*. http://www.omg.org/spec/UML/2.3/. – Stand 26.6.2011

[OSG] OSGI ALLIANCE: *OSGi Service Platform Release 4.0*. http://www.osgi.org/Specifications/HomePage. – Stand 26.6.2011

[OW07] ORAM, Andy ; WILSON, Greg: *Beautiful Code: Leading Programmers Explain How They Think*. 1. Auflage. O'Reilly Media, 2007. – ISBN 9780596510046

[Par07] PARET, Dominique: *Multiplexed Networks for Embedded Systems: CAN, LIN, Flexray, Safe-by-wire...* SAE International, 2007. – ISBN 9780768019384

[Rei08] REIF, Konrad: *Automobilelektronik: Eine Einführung für Ingenieure*. 3. Auflage. Vieweg+Teubner, 2008. – ISBN 9783834804464

[Sam08] SAMEK, Miro: *Practical UML Statecharts in C/C++: Event-Driven Programming for Embedded Systems*. 2. Auflage. Newnes, 2008. – ISBN 9780750687065

[Sar11] SARADJUK, Markus: *Entwicklung eines intelligenten Systems zur Aufzeichnung von LIN-Kommunikation*. 2011. – Bachelor Thesis, Hochschule für Technik Stuttgart

[Str94] STROUSTRUP, Bjarne: *The Design and Evolution of C++*. 1. Auflage. Addison-Wesley Professional, 1994. – ISBN 9780201543308

[TMD09] TAYLOR, R.N. ; MEDVIDOVIC, N. ; DASHOFY, E.M.: *Software Architecture: Foundations, Theory and Practice*. 1. Auflage. Wiley, 2009. – ISBN 9780470167748

[Vec] VECTOR INFORMATIK GMBH: *Dokumentation zur Vector XL Library*. – Stand 26.6.2011

[Vli] VLISSIDES, John: *Generation Gap Pattern*. http://www.research.ibm.com/designpatterns/pub/gg.html. – Stand 26.6.2011

[XS05] XING, Zhenchang ; STROULIA, Eleni: *UMLDiff: An Algorithm for Object-Oriented Design Differencing*. 2005

[ZS10] ZIMMERMANN, Werner ; SCHMIDGALL, Ralf: *Bussysteme in der Fahrzeugtechnik: Protokolle, Standards und Softwarearchitektur*. 4. Auflage. Vieweg+Teubner, 2010. – ISBN 9783834809070