Methodischer Beitrag zum neuen Einsatz von Techniken der formalen Verifikation bei seriellen Busprotokollen im automobilen Umfeld

Zur Erlangung des akademischen Grades eines

DOKTOR-INGENIEURS

von der Fakultät für
Elektrotechnik und Informationstechnik
des Karlsruher Instituts für Technologie (KIT)
genehmigte

DISSERTATION

von

Dipl.-Ing. Jens Eckart Becker

geboren in Karlsruhe

Tag der mündlichen Prüfung:
25.11.2014

Hauptreferent: Prof. Dr.-Ing. Dr. h. c. Jürgen Becker
Korreferent: Prof. Dr.-Ing. Norbert Wehn (TU Kaiserslautern)

Methodischer Beitrag zum neuen Einsatz von Techniken der formalen Verifikation bei seriellen Busprotokollen im automobilen Umfeld

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Der Trend bei modernen Automobilen geht dazu, dass ein immer größerer Anteil der Innovationen in der Elektronik liegen. Moderne Autos bieten inzwischen eine Vielzahl von Fahrerassistenzsystemen wie zum Beispiel automatische Abstandsregelung, Spurhalteassistenten, Nachtsichtkameras zur Erkennung von Personen am Straßenrand und viele andere mehr. Einher mit diesen immer komplexeren elektronischen Systemen geht auch eine Zunahme von im Automobil verbauten Steuergeräten. Während diese Steuergeräte am Anfang der Entwicklung noch isoliert und unabhängig voneinander funktionierten, ist zur Ermöglichung der aktuellen Funktionalitäten eine Kommunikation einzelner Steuergeräte (Electronic Control Unit (ECU)) untereinander nötig.

Während dazu früher proprietäre Systeme zum Einsatz gekommen sind, werden heutzutage hauptsächlich standardisierte Kommunikationssysteme wie zum Beispiel Controller Area Network (CAN), LIN, Media Oriented Systems Transport (MOST) und FlexRay verwendet. Dies ist besonders wichtig vor dem Hintergrund einer möglichst großen Wiederverwendbarkeit sowie Flexibilität beim Einsatz verschiedener Steuergeräte. Aktuelle Steuergeräte werden oft nicht mehr direkt beim Automobilhersteller (Original Equipment Manufacturer (OEM)) entwickelt und gebaut, sondern nach Spezifikation von verschiedenen Zulieferern gefertigt. Das bedeutet, dass die einzelnen Komponenten völlig unabhängig von einander entwickelt und getestet werden und erst bei der Integration beim OEM zum ersten Mal zusammen arbeiten müssen. So kann es vorkommen, dass Inkonsistenzen bei der Umsetzung der Kommunikationsprotokolle erst bei der Integration entdeckt werden. Die Behebung dieser Fehler kann einen erheblichen Verlust an Zeit und Geld bedeuten. Um eine einwandfreie und fehlerlose Integration der einzelnen, bei verschiedenen Herstellern gefertigten Komponenten zu ermögli-

chen, ist es deshalb zwingend notwendig, dass das gemeinsame Kommunikationsprotokoll von allen Komponenten gleich implementiert wird.

Zur Überprüfung der Einhaltung der geforderten Spezifikationen existieren verschiedene Ansätze wie Simulation , HiL, Rapid Prototyping und formale Verifikation. Diese Ansätze sollen hier nur kurz vorgestellt werden. Weitere Informationen finden sich in Kapitel 2.

Bei der Simulation wird das zu testende Steuergerät in ein abstrahiertes Modell abgebildet. Damit ist eine Ausführung des Modells auf einem Rechner möglich. Um das Verhalten des Modells zu testen, werden die Eingänge mit vordefinierten und/oder zufällig erzeugten Testvektoren beaufschlagt. Die Antwort des Modells auf diese Testvektoren wird aufgezeichnet und kann auf Übereinstimmung mit den geforderten Reaktionen geprüft werden (Conformance Test). Bedingt durch das Grundprinzip der Simulation gibt es Einschränkungen bezüglich deren Aussagekraft. So kann eine Simulation im Allgemeinen nur Aussagen über das Vorhandensein von Fehlern, nicht aber zu deren Abwesenheit treffen. Für definitive Aussagen müssten alle möglichen Eingangskombinaten eines Systems in jeder zeitlichen Abfolge überprüft werden. Bei einer kleinen Anzahl von Eingangssignalen und wenigen internen Zuständen ist das noch durchführbar. Mit dem zunehmenden zu betrachtenden Raum bei mehr Signalen und Zuständen wird die für die Simulation benötigte Zeit jedoch zu einem limitierenden Faktor. Deswegen muss man sich im praktischen Einsatz auf eine begrenzte Zahl von Testvektoren beschränken, so dass deren Auswahl besondere Bedeutung für die Qualität des Simulationsergebnisses zukommt. Zu Situationen, für die kein Testvektor existiert, kann damit keine definitive Aussage über das Verhalten des Systems getroffen werden.

Beim HiL wird das reale Steuergerät in einer simulierten Umgebung getestet. Diese Tests können damit erst in einem späten Stadium der Entwicklung durchgeführt werden, wenn das Steuergerät vorhanden ist. Es eignet sich also weniger, um frühzeitig Fehler im Design aufdecken und beheben zu können. Die Anzahl und die Laufzeit der überprüften Testfälle ist zudem durch die Geschwindigkeit des zu testenden Gerätes beschränkt. Es ist damit im Normalfall nicht möglich, die Fehlerfreiheit eines Gerätes für alle Fälle nachzuweisen.

Einen anderen Ansatz verfolgt das Rapid Prototyping. Hierbei geht es darum, das zu entwickelnde Steuergerät möglichst früh in der realen Umgebung zu testen. Da das endgültige Hardwaredesign noch nicht zur Verfü-

gung steht, kommen dabei zum Beispiel Field Programmable Gate Arrays (FPGAs) zum Einsatz. Mit ihnen ist es möglich, das Steuergerät mit der geplanten Hardware-Architektur zu testen, ohne das diese schon zur Verfügung steht. Ein Nachteil dieser Nachbildung der Hardware ist jedoch häufig, dass die Ausführungsgeschwindigkeiten der Vorstufe nicht mit denen der späteren Hardware übereinstimmen. Damit ist ein Test von Echtzeitanforderungen nicht möglich. Ähnlich wie bei der Simulation und HiL besteht auch hier das Problem des großen Zustandsraumes von realen Anwendungen. Ein vollständiger Test aller möglichen Zustände ist in der zur Verfügung stehenden Zeit nicht möglich.

Einen anderen Ansatz stellt die Verifikation eines Systems mit Hilfe der formalen Verifikation dar. Hierzu wird das zu betrachtende System in ein mathematisches Modell überführt. Damit ist es möglich nachzuweisen, dass eine zu überprüfende Eigenschaft für alle möglichen Ausführungen des Modells gilt. Es ist also nicht nötig, einzelne Testszenarien zu generieren. Damit werden sämtliche möglichen Randfälle automatisch mit berücksichtigt. Nachteilig bei der formalen Verifikation ist jedoch im Allgemeinen deren Aufwand und enge Grenzen bezüglich der Komplexität der zu untersuchenden Modelle. So wächst der zu betrachtende Zustandsraum exponentiell mit der Anzahl der Eingänge und der internen Zustände eines Systems. Die Überprüfung eines größeren Systems ist damit in der oftmals nur begrenzt zur Verfügung stehenden Zeit nicht möglich.

Um dieses Problem zu beherrschen, beschränken sich aktuell im industriellen Umfeld eingesetzte Verfahren wie das Bounded Model Checking (BMC) [3] oder das Interval Property Checking (IPC) [18] auf eng begrenzte Zeitfenster, in denen das Verhalten des Systems untersucht wird. Dazu wird das Modell nur für die Anzahl von Zeitschritten untersucht, für die die zu betrachtende Eigenschaft Gültigkeit besitzen soll. Der Aufwand für die Verifikation wird also vom gesamten Zustandsraum auf das deutlich kleinere Zeitfenster reduziert.

Für die Verifikation der Befehlsausführung von Prozessoren können formale Methoden heute schon sehr effizient eingesetzt werden [5]. Ein Grund dafür ist, dass die einzelnen Befehle in ihrer Ausführung kaum von der Historie des Systems abhängig sind. Auch die Ausführung selbst benötigt nur wenige Taktzyklen. Damit ergibt sich ein relativ kleines Zeitfenster, dass für die formale Verifikation betrachtet werden muss. Anders stellt sich die Situation allerdings bei der Kommunikation dar. Hier ist das zu betrachtende

Zeitfenster bedeutend größer, vor allem bei seriellen Kommunikationsprotokollen. Im Gegensatz zu paralleler Kommunikation, bei der Daten und Steuerbefehle oft gleichzeitig oder zeitlich nahe beieinander vorliegen, werden diese Informationen bei der seriellen Kommunikation nacheinander übertragen. Ein Übertragungszyklus erstreckt sich also nicht auf wenige Taktzyklen, sondern kann durchaus einige Hundert bis Tausende Taktzyklen dauern. Die sich daraus ergebenden großen zu betrachtenden Zeitfenster stellen eine besondere Herausforderung für die formale Verifikation dar. Sie können zu sehr langen Laufzeiten für die Verifikation führen oder den Nachweis sogar auf Grund der zu großen Komplexität unmöglich machen.

1.2 Beiträge der Dissertation

Die vorliegende Arbeit liefert einen methodischen Beitrag zum Einsatz von State of the Art Techniken der formalen Verifikation speziell für die Verwendung bei seriellen Kommunikationsprotokollen aus dem automobilen Umfeld.

- Eine Divide et Impera Methodik zur Zerlegung des Protokolls in kleine, eigenständige Abschnitte, die mit aktuellen Verifikationstools in überschaubarer Zeit verifiziert werden können. Durch die Aneinanderkettung der einzelnen Abschnitte kann so das gesamte Protokoll vollständig beschrieben werden. Der saubere Übergang von einem Abschnitt zum nächsten wird dabei über eine geeignete Darstellung der Annahmen und des Beweisteils erreicht. Mit Hilfe dieser Methodik ist es auch möglich, Übertragungen zu Verifizieren, bei denen zwischen einer Anfrage (Request) und der zugehörigen Antwort (Response) beliebige Zeiträume liegen können.
- Bisherige Arbeiten auf dem Gebiet der Verifikation von Kommunikationsprotokollen beschränken sich auf den Nachweis des korrekten Verhaltens auf dem Bus selbst. In Ergänzung dazu betrachtet die in dieser Arbeit vorgestellte Methodik auch die korrekte Datenverarbeitung innerhalb des Moduls und deren Weitergabe an externe Komponenten. Da hierzu auf Interna der Implementierung zurück gegriffen werden muss, sind die so erstellten Properties implementierungsspezifisch. Sie lassen sich dadurch nur schwer bis gar nicht für die Verifikation einer anderen Implementierung der gleichen Funktionalität verwenden. Die

vorliegende Arbeit stellt einen Ansatz zur Abstraktion der verwendeten Properties von der aktuellen Implementierung vor.

1.3 Struktur der Dissertation

In Kapitel 2 werden kurz die Grundlagen der formalen Verifikation vorgestellt. Es handelt sich dabei nicht um eine mathematische Abhandlung der dahinter liegenden Theorie, sondern fasst die für das Verständnis dieser Arbeit notwendigen Punkte zusammen. Ein weiterer Teil dieses Grundlagenkapitels ist die kurze Beschreibung ausgewählter serieller Busprotokolle im automobilen Umfeld mit besonderem Augenmerk auf dem in dieser Arbeit untersuchten LIN Protokoll.

Im Kapitel 3 werden die besonderen Herausforderungen bei der Verifikation serieller Protokolle dargestellt. Außerdem werden hier existierende Ansätze zur Verifikation serieller Protokolle vorgestellt und von der vorliegenden Arbeit abgegrenzt.

Das folgende Kapitel 4 beschreibt die neue Methodik zur Verifikation serieller Protokolle. Neben der effizienten Behandlung langer serieller Protokolle ermöglicht sie auch eine einfache Abstraktion von einer konkreten Implementierung eines Protokolls durch die Einführung einer Mappingschicht. Dieser Ansatz ermöglicht es, dass ein einmal erstellter Satz von Eigenschaftsbeschreibungen nahezu unverändert auf verschiedenen Implementierungen angewandt werden kann.

Die Anwendung der Methodik auf ein konkretes Fallbeispiel wird danach in Kapitel 5 anhand eines industriell eingesetzten LIN-Cores gezeigt.

Den Abschluss der Arbeit bildet Kapitel 6, in dem die wesentlichen Ergebnisse zusammengefasst und ein Ausblick auf mögliche weitere Arbeiten gegeben wird.

2 Grundlagen

2.1 Systementwurf komplexer elektronische Systeme

Die Komplexität der im Automobil eingesetzten elektronischen Systeme hat seit Beginn der Geschichte stetig zugenommen. War in der Anfangszeit nur die Zündung für den Motor elektrisch betrieben, so kamen im Laufe der Jahre weitere Funktionen wie elektrische Beleuchtung und Anlasser hinzu. Diese wurden im Laufe der Zeit ergänzt durch weitere Anwendungen wie zum Beispiel Scheibenwischer, Radio und Fensterheben. Zu Beginn handelte es sich dabei noch um von einander unabhängige Geräte. Im Laufe der Weiterentwicklung wurden diese Einzelkomponenten jedoch immer mehr zu einen Gesamtsystem integriert. Erst diese Integration ermöglichte komplexe Funktionen wie zum Beispiel das Antiblockiersystem, bei dem die Informationen von Umdrehungssensoren an den einzelnen Rädern genutzt werden, um das Auto in kritischen Fahrsituationen durch gezielten Bremseingriff stabilisieren zu können. Damit werden die vorher getrennten Funktionalitäten „Geschwindigkeitsmessung“ und „Bremsen“ zu zu einer neuen Funktionalität „Fahrstabilisierung“ verknüpft. Waren es anfangs nur einzelne Funktionalitäten, die miteinander kombiniert wurden, so steigert sich der Integrationsgrad der Fahrzeugelektronik in den letzten Jahren immer weiter.

Ein solch komplexes System kann nicht mehr als ein einzelner Block entworfen und entwickelt werden. Es erfordert vielmehr eine Aufteilung der Gesamtfunktionalität in einzelne, überschaubare Teilsysteme, die später zu einem Ganzen zusammen gesetzt werden. Ein zur Beschreibung und Durchführung dieses Prozesses häufig verwendetes Modell stellt das so genannte V-Modell dar. Aus dem allgemeinen Vorgehensmodell in der Softwareentwicklung, dass zuerst vom US-amerikanischen Softwareingenieur Barry Boehm im Jahr 1979 vorgeschlagen wurde, entwickelte sich später auch ein Entwicklungsstandard der öffentlichen Hand in Deutschland mit gleichem Namen [11]. Neben der Anwendung bei der Entwicklung von Software-

Projekten kommt es auch mit leichten Abweichungen bei der Entwicklung von Hardware- und gemischten Hardware/Software-Systemen zum Einsatz.

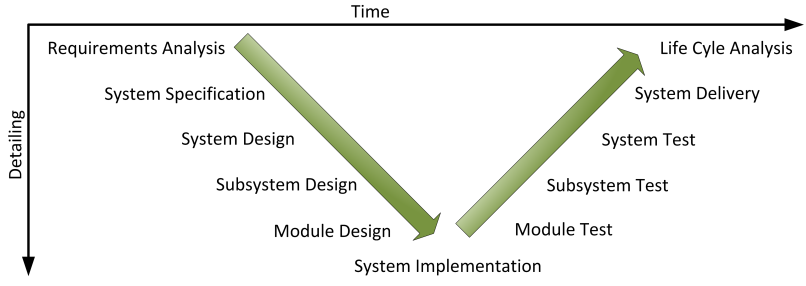


Abbildung 2.1: V-Modell

Kern des V-Modells ist der phasenweise Entwicklungsprozess. Dabei wird das System von Phase zu Phase in einem Top-Down-Ansatz immer weiter detailliert und seine Funktionalität genauer festgelegt. Aus einem einzigen Gesamtsystem entstehen so zum Beispiel einzelne funktionale Blöcke die im Anschluss an die Planungs- und Spezifikationsphasen implementiert werden. Dieser Teil des Entwicklungsprozesses ist im linken, absteigenden Ast der graphischen Darstellung des V-Modells (vergl. Abbildung 2.1) zu sehen. Von der Festlegung der generellen Anforderungen an das zu entwickelnde Systems (oben links) reicht dieser Ast bis zur Implementierung der spezifizierten Subsysteme (unten Mitte). Der Ergebnisse einer Phase sind dabei bindende Vorgaben für die in der Detaillierung darunter liegende Projektphase.

Im Gegensatz zu anderen Modellen, wie zum Beispiel dem Wasserfallmodell ([19]), wird beim V-Modell zusätzlich zu dem Vorgehen zur Entwicklung auch das Vorgehen zur Qualitätssicherung (Testen) definiert. Diesem ist der rechte, aufsteigende Ast des V-Modells zugeordnet. In diesem Ast werden die einzelnen erstellten Komponenten in immer komplexerer Zusammenfassung getestet. Dabei steht jeder Phase auf der Seite der Spezifikation auch immer genau eine Phase auf der Test-Seite gegenüber, so dass sich in der graphischen Darstellung das namensgebende „V“ ergibt. Mit dieser direkten Verknüpfung von Design- und Testphasen soll eine möglichst umfassende Testabdeckung des Gesamtsystems erreicht werden.

2.2 Verifikation von Hardware

Wie in Kapitel 2.6 genauer ausgeführt, werden heutzutage im automobilen Umfeld immer mehr standardisierte Bussysteme für die Kommunikation zwischen den einzelnen Steuergeräten eingesetzt. Die Verwendung standardisierter Schnittstellen und Protokolle ermöglicht es, Komponenten unterschiedlicher Hersteller miteinander zu verbinden, ohne dass Anpassungen an einzelnen Komponenten vorgenommen werden müssen. Im Idealfall kann die Funktionalität eines Gesamtsystems dabei durch einfaches Hinzufügen oder Austauschen einer Komponente ergänzt oder verbessert werden.

Hierfür müssen jedoch weitere Voraussetzungen eingehalten werden, die üblicherweise in Form einer Spezifikation festgelegt werden. Diese Spezifikation besteht aus mehreren Teilen. Unter anderem sollten dort Vorgaben zu mechanischen Eigenschaften, wie zum Beispiel zu zulässigen Steckverbindern gemacht werden. Außerdem muss dort festgelegt werden, welche elektrischen Eigenschaft wie zum Beispiel die zu verwendenden Spannungspegel oder die Flankensteilheit der Signale, einzuhalten sind. Diese beiden Aspekte spielen jedoch im Rahmen der hier vorliegenden Arbeit keine Rolle.

Wichtig hingegen sind logische und zeitliche Aspekte ab der Bitebene, die ebenfalls in einer vollständigen Spezifikation festgelegt werden müssen. So werden dort Festlegungen zu den möglichen Rahmenformaten und deren Verwendung getroffen. Die Definition der Rahmenformate ermöglicht es, einzelnen empfangenen Bits ihre jeweilige Bedeutung zuzuordnen und damit zum Beispiel empfangene Daten von Steuerinformationen zu trennen. Nur wenn alle an der Kommunikation beteiligten Knoten gesendeten Bits in gleicher Weise interpretieren, ist eine fehlerfreie Kommunikation möglich. Weiterhin muss eine Spezifikation Vorgaben zum korrekten Einsatz der Rahmenformate machen. Dadurch wird zum Beispiel festgelegt, mit welchem Rahmen ein Übertragung eingeleitet wird und wie ein anderer Knoten darauf zu reagieren hat.

Eine fehlerfreie Kommunikation ist also nur möglich, wenn sich alle beteiligten Knoten exakt an die in der Spezifikation festgelegten Regeln halten. Nur so ist es möglich, einzelne Teile unabhängig voneinander zu entwickeln und dann am Ende zu einer funktionierenden Einheit zusammen zu fassen. Durch das Zusammenspiel von verschiedenen Komponenten ergeben sich unter Umständen völlig neue Fehlerfälle, die so im Einzelbetrieb nicht aufgetaucht sind. Da diese Fehler durch das wechselseitige Zusammenspiel der

einzelnen Komponenten entstehen, sind sie nur schwer zu lokalisieren und zu beheben. Um solche Fehler nach Möglichkeit ganz auszuschließen, ist es wünschenswert, für jedes Steuergerät einzeln nachweisen zu können, dass die Spezifikationen erfüllt werden. In der Theorie können Fehler beim Zusammenschluss dann nur noch daher rühren, dass die Spezifikation selbst fehlerhaft ist.

Bei der Spezifikation für ein solches vernetztes Steuergerät lassen sich dabei grob zwei verschiedenen Teile unterscheiden. Da ist zum einen die Spezifikation der Funktionalität selbst. Damit wird festgelegt, wie das Steuergerät auf eingehende Daten reagiert und welche Aktionen und Ausgaben es produziert. Zusätzlich enthält sie auch weitere physikalische Festlegungen wie die verwendeten Spannungen und Ströme, die Temperaturbereiche, Abmessungen usw. Dieser Teil der Spezifikationen wird in dieser Arbeit nicht betrachtet. Der zweite Teil der Spezifikation ist die Definition des Kommunikationsprotokolls auf dem gemeinsam genutzten Bus. Da das Kommunikationsprotokoll die Grundlage für den Austausch von Daten zwischen den einzelnen Steuergeräten darstellt, ist dessen korrekte Implementierung essentiell für das Funktionieren des Gesamtsystems.

Der frühen Entdeckung und Behebung von Fehlern kommt im Automobilbereich eine besondere Bedeutung zu. Hier ist es inzwischen üblich, dass sich ein System aus Komponenten von verschiedenen Zulieferern zusammensetzt, die wiederum Teilkomponenten von ihren Zulieferern erhalten. Das Verhalten des Gesamtsystems, dass beim Original Equipment Manufacturer (OEM) zusammengesetzt wird, ist damit von dem korrekten Verhalten von Teilkomponenten abhängig, über die der OEM keine direkte Kontrolle mehr hat. Besonders bei den direkt an der Verarbeitung des Kommunikationsprotokolls beteiligten Komponenten kann sich ein Fehlverhalten direkt negativ auf die Funktion des Gesamtsystems auswirken. Zur Beseitigung dieser Fehler kann, wenn er erst bei der Integration entdeckt wird, ein erheblicher Aufwand an Zeit und Kosten erforderlich sein. Aus diesem Grund ist es für den OEM erstrebenswert, die korrekte Funktionalität schon früh im Prozess mit den Teilkomponenten sicher zu stellen.

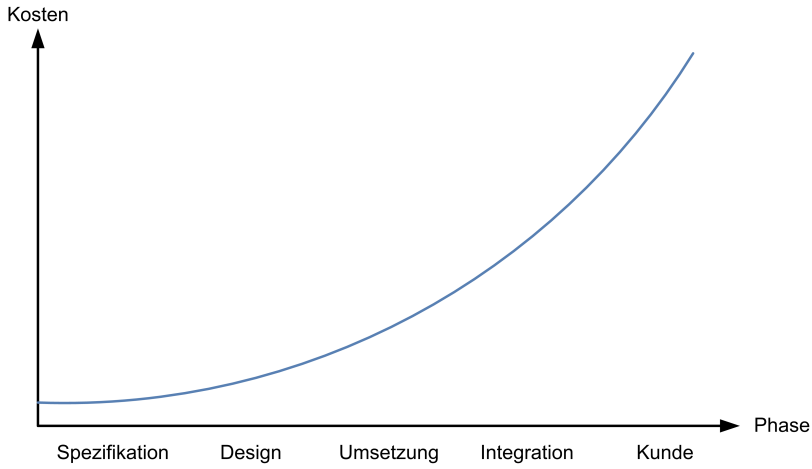


Abbildung 2.2: Kosten zur Behebung von Fehlern in verschiedenen Phasen der Entwicklung [13]

2.3 Ansätze zur Verifikation von Hardware

Für die Verifikation von Hardware-Komponenten existieren verschiedene Ansätze mit unterschiedlichen Stärken und Schwächen. Diese sollen in dem folgenden Abschnitt kurz vorgestellt werden.

2.3.1 Simulation

Unter Simulation versteht man die Nachbildung des Verhaltens eines realen Systems durch Modelle. Diese Modelle können dann in einem Rechner mit verschiedenen Eingangsgrößen beaufschlagt werden, um das Verhalten des Systems auf diese Stimuli zu untersuchen. Damit ist es beim Systementwurf möglich, die funktionalen Eigenschaften eines Systems mit den in einer Spezifikation geforderten Eigenschaften zu vergleichen. Eine Abweichung von den geforderten Eigenschaften wird als Fehler des Systems angesehen. Hält

ein System alle in der Spezifikation geforderten Eigenschaften ein, gilt es als fehlerfrei.

Je nachdem, wie genau diese Modelle das reale System abbilden, unterscheidet man verschiedene Abstraktionsebenen. Die bei der Modellierung von elektronischen Systeme gebräuchlichen Abstraktionsebenen sind in Abbildung 2.3 zu sehen.

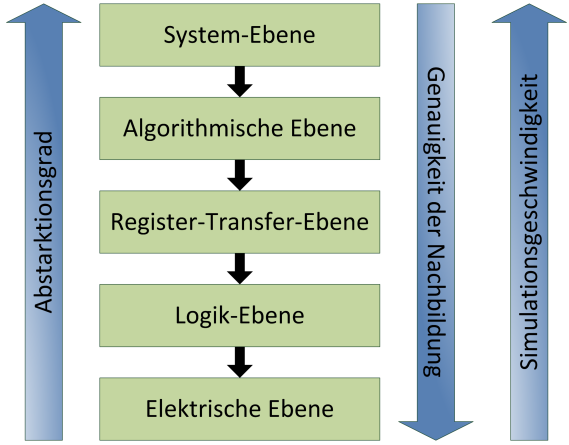


Abbildung 2.3: Abstraktionsebenen bei der Modellbildung

Von der Systemebene hin zur elektrischen Ebene steigt die Genauigkeit des Modells. Es bildet also das reale System immer genauer ab. Gleichzeitig nimmt jedoch die Komplexität der Modell zu. Diese steigende Komplexität führt im Allgemeinen dazu, dass die Zeit zur Ausführung eines Simulationsmodells ansteigt. Hier ist also eine Abwägung zwischen der zu erzielenden Genauigkeit der Nachbildung und dem dafür benötigten Aufwand erforderlich. Während die Modelle auf den hohen Abstraktionsebenen üblicherweise deutlich schneller ausgeführt werden können, als das beim realen System der Fall ist, so ist es auf den niedrigen Ebenen häufig der Fall, dass die Dauer der Simulation erheblich größer ist, als das zu simulierende reale Geschehen.

Bei der Simulation kommen sowohl für das zu verifizierende Steuergerät als auch für die Umgebung, mit der es interagieren soll, nur Modell zur

Anwendung. Dies ermöglicht zumindest in der Theorie beliebige Ausführungsgeschwindigkeiten für die Simulation, solange die verwendeten Rechner leistungsfähig genug sind. Damit ist eine beschleunigte Verifikation im Vergleich zur Durchführung mit realen Steuergeräten und in realer Umgebung möglich. Zudem können so auch Szenarien untersucht werden, die in der Realität nur schwer zu erreichen sind, weil sie zum Beispiel selten auftreten oder gefährlich sind und zu einer Beschädigung oder Zerstörung führen können. Außerdem ermöglicht die Simulation beliebig häufige, exakte Wiederholung von Szenarien. Dies erleichtert die iterative Verbesserung des zu verifizierenden Gerätes, da sich die Auswirkungen einer Änderung unter gleichen Bedingungen exakt nachvollziehen lassen.

Auf der anderen Seite hingegen besteht die Gefahr, dass das zu verifizierend Gerät und/ oder die Umgebung nicht ausreichend genau in dem Modell erfasst wurden. Ein Gerät, das in der Simulation funktioniert, muss deswegen nicht unbedingt auch in der Realität funktionieren oder zumindest das gleiche Verhalten zeigen. Damit können mögliche Fehler unentdeckt bleiben. Um die Realität besser abbilden zu können, müssen die Modelle verfeinert werden, was wiederum zu längeren Laufzeiten führt. Jedoch ist hier zu beachten, dass eine Simulation im Allgemeinen nur Fehler im betrachteten System aufdecken kann. Der Nachweis der Fehlerfreiheit ist auf Grund der Komplexität der zu untersuchenden Systeme mit der Simulation nicht zu erreichen.

Statt dessen kann immer nur das korrekte Verhalten in genau dem untersuchten Szenario gezeigt werden. Kleine Abweichungen von diesem Szenario können jedoch schon wieder einen Fehler erzeugen. Dies rührt unter anderem daher, dass es bei größeren Modellen praktisch unmöglich ist, alle möglichen Eingangsbelegungen bei allen möglichen internen Zuständen zu simulieren. Hier wird aus Gründen der Praktikabilität also nur eine Auswahl aller möglichen Szenarien simuliert werden.

Eine alleinige manuelle Bestimmung der zu simulierenden Szenarien ist darauf beschränkt, was der Verifikateur für zu simulieren notwendig erachtet. Dabei besteht die Gefahr, dass Randfälle und ungewöhnliche Konstellationen unbeachtet bleiben, weil der Verifikateur sie nicht in Betracht gezogen hat. Aus diesem Grund werden die manuell vorgegebenen Szenarien üblicherweise durch weitere, maschinell erzeugte Szenarien ergänzt. Geschieht dies rein zufällig, so spricht man von „random testing“. Dieses Verfahren liefert zwar auf einfache Weise zusätzliche Testfälle, die möglicherweise wei-

tere Fehler aufdecken. Allerdings ist mit diesem Ansatz eine hohe Anzahl von Testfällen nötig, um eine breite Abdeckung aller möglichen Szenarien zu erreichen. Die so genannten „direkted random simulation“ kann als Kombination aus den beiden vorher genannten Ansätzen angesehen werden. Auch dabei werden die einzelnen Testfälle zufällig erzeugt. Allerdings erfolgt diese Erzeugung nach den Vorgaben, die der Verifikateur gemacht hat, um zu erreichen, dass kritische Teile des System stärker getestet werden. Auch Randfälle wie minimale und maximale Werte lassen sich damit extensiver testen. Während sich die Abdeckung der Simulation durch die Hinzunahme von weiteren Szenarien erhöhen lässt, ist eine vollständige Abdeckung in der Realität aber kaum erreichbar.

2.3.2 Hardware in the Loop (HiL)

Während bei der Simulation sowohl das zu verifizierende Steuergerät als auch die Umgebung nur als Simulationsmodell vorliegen, handelt es sich beim Hardware in the Loop (HiL) um eine Kombination aus simulierter Umgebung und realem Steuergerät. Dazu wird ein reales Steuergerät mit seinen Ein- und Ausgängen mit einem so genannten HiL-Simulator verbunden. Dieser Simulator dient der Nachbildung der Umgebung, in der das zu verifizierende Steuergerät eingesetzt werden soll. Eine Prinzipskizze dazu ist in 2.4 zu sehen.

Der HiL-Simulator besteht aus mindestens zwei verschiedenen Komponenten. Eine Komponente ist ein echtzeitfähiger Rechner, der für die Ausführung des Simulationsmodells und die Steuerung des Gesamtsystems zuständig ist. Die in diesem Rechner zur Verfügung stehende Rechenleistung beeinflusst maßgeblich die Genauigkeit der simulierten Umgebung, da die Reaktion der Simulation auf die Ausgänge des zu verifizierenden Steuergerätes praktisch in Realzeit verfügbar sein müssen.

Zur Klärung der hier verwendeten Bedeutung des Begriffs „Realzeit“, soll zuerst der Begriff „Echtzeit“ definiert werden.

Definition 2.1 (Echtzeit)**.** Vorgegebene Zeit, die bestimmte Prozesse einer elektronischen Rechenanlage in der Realität verbrauchen dürfen. (aus [7])

„Ein Gerät ist echtzeitfähig“ bedeutet also, dass die maximale Antwortzeit eines Systems festgelegt ist. Aussagen über die Dauer dieser Antwortzeit

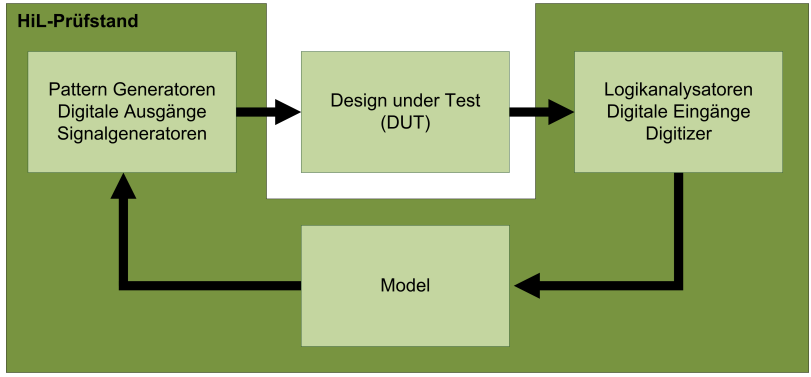


Abbildung 2.4: Prinzip des HiL

und besonders ihrer Beziehung zu realen Vorgängen werden damit jedoch nicht getroffen. Somit ist ein System immer dann echtzeitfähig, wenn seine Antwort sicher innerhalb einer festgelegten Zeit erfolgt, auch wenn die Zeit zu Beispiel mehrere Stunden beträgt. Damit ist diese Definition jedoch nicht ausreichend zur Beschreibung eine HiL-Systems, da hier ein fester Zusammenhang zum realen Zeitablauf besteht. Zur genauen Unterscheidung soll hier die „Realzeit“ definiert werden:

Definition 2.2 (realzeitfähig)**.** Ein System ist dann realzeitfähig, wenn es echtzeitfähig ist und seine Reaktionen immer synchron zu einem realen Prozess ablaufen.

In der Praxis bedeutet das, dass ein System dann realzeitfähig ist, wenn seine Reaktionsgeschwindigkeit so groß ist, dass es immer rechtzeitig auf relevante Änderungen der realen Umgebung reagieren kann.

Die zweite essentielle Komponente des HiL-Systems sind die Ein- und Ausgabeschnittstellen, die die reale Hardware mit der Simulation im Rechner verknüpfen. Diese Schnittstellen haben zum einen die Aufgabe, die Ausgangssignale des zu verifizierenden Steuergerätes in eine digitale Darstellung umzuwandeln und zum anderem die errechneten Größen der simulierten Umgebung als reale Analogsignale für das Steuergerät bereit zu stellen.

Solange es sich bei dem zu testenden Gerät nur um das reine Steuergerät handelt, sind diese Schnittstellen relativ einfach aufgebaut, da darüber im wesentlichen nur Signale von kleiner Spannung und Stromstärke ausgetauscht werden müssen. Schon bei der Integration von Leistungstreibern in das Steuergerät nimmt der Aufwand bei der Ausgestaltung der Schnittstellen deutlich zu, da diese nun mit größeren Stromstärken und Spannungen arbeiten müssen. Eine weitere Steigerung der Komplexität ergibt sich, wenn das Steuergerät nicht mehr rein elektrisch mit der Umgebung kommuniziert, sondern zum Beispiel schon Sensoren und Aktuatoren direkt integriert sind. Diese müssen dann unter Umständen mechanisch angesprochen werden, was einen zusätzlichen Aufwand für die Bereitstellung der nötigen Schnittstellen bedeutet.

Aufgrund der oben beschriebenen Komplexität zur Simulation der Umgebung und der Tatsache, dass das zu verifizierende Steuergerät schon in Hardware vorliegen muss, findet HiL erst in den späteren Phasen der Entwicklung statt. Der Haupteinsatzzweck von HiL ist damit nicht die Verifikation des Designs und der Implementierung in frühen Phasen der Entwicklung. Stattdessen liegt der Schwerpunkt darauf, Tests in der Phase der Systemintegration vornehmen zu können. Hier ist es möglich, das fast fertig entwickelte Steuergerät in realistischen Umgebungen testen zu können, ohne das diese wirklich vorhanden sein müssen. HiL hat hier einige Vorteile im Vergleich zu einem Test in realen Umgebungen. So ist man ganz praktisch nicht auf das Vorhandensein einer geeigneten Umgebung angewiesen. Der Test kann an nahezu jedem beliebigen Ort stattfinden, an dem der HiL-Prüfstand verfügbar ist. Aus der Unabhängigkeit von realen Bedingungen ergibt sich auch eine sehr gute Wiederholbarkeit der durchgeführten Tests. Ein Steuergerät kann damit immer wieder unter praktisch gleichen Bedingungen getestet werden. Dies ist besonders wichtig beim iterativen Vorgehen zur Fehlerbeseitigung oder Optimierung. Diese Eigenschaften sind auch förderlich für die Arbeit in verteilten Teams, bei denen mehrere Entwickler und Verifikateure an unterschiedlichen Orten und zu unterschiedlichen Zeiten zusammenarbeiten sollen. Ein weiterer Vorteil des Testens mit HiL ist die Möglichkeit, auch Tests durchführen zu können, die in der Realität gefährlich wären oder zur Beschädigung oder Zerstörung des Steuergerätes und/oder der Umgebung führen könnten. Eine Wiederholung von solchen Tests würde aufgrund der Beschädigung zu hohen Kosten führen. Beim HiL Test handelt es sich dagegen nur um etwas andere Parameter, die an das Steuergerät weitergegeben werden.

Eine deutliche Einschränkung bei HiL stellen jedoch die Schnittstellen dar, die benötigt werden, um das zu testende Steuergerät mit dem simulierten Modell zu verbinden. Im Gegensatz zur Simulation, bei der nur Daten innerhalb des Simulationsmodells im Rechner weitergegeben werden müssen, ist es bei HiL erforderlich, reale physikalische Signale zu erzeugen, die vom Steuergerät verarbeitet werden können. Digitalsignale zur Anbindung an die IO-Schnittstellen sind dabei relativ einfach zu erzeugen. Jedoch sind auch hier zusätzliche Kriterien zu erfüllen. So muss zum Beispiel die Möglichkeit gegeben sein, auch Fehlerfälle auf den Steuersignalen nachzubilden. Dies können zum Beispiel feste Signalwerte genauso wie Kurzschlüssel oder offene Leitungen sein. Zu deren Nachbildung sind für jedes einzelne Signal aufwendige Schaltungen vorzusehen.

Neben diesen Niederspannungs- und Niederstromsignalen besteht jedoch auch im Allgemeinen die Notwendigkeit, zusätzliche Ein- und Ausgänge bereit zu stellen, die höhere Spannungen oder Ströme liefern oder aufnehmen können. Für den Fall, dass im Steuergerät Sensorik integriert ist, die zum Beispiel direkt einen Motorstrom messen kann, muss dieser auch im HiL-Prüfstand in der benötigten Stärke breit gestellt werden. Ähnlich ist die Situation auch bei den Eingangsgrößen des HiL-Prüfstandes. Hier ist es zum Beispiel nicht ausreichend, nur ohmsche Lasten zur Verfügung zu stellen. Für die Nachbildung eines Elektromagneten werden zum Beispiel auch induktive Lasten benötigt. Mit der Nachbildung dieser Ein- und Ausgangsgrößen bestimmt sich auch direkt die Qualität der Testergebnisse des Prüfstandes. Sind diese nicht korrekt den Originalgrößen nachgebildet, stimmt die Reaktion des getesteten Steuergerätes im Prüfstand unter Umständen nicht mit der Reaktion in der realen Umgebung überein. Umgekehrt muss der Prüfstand aber keine Signale liefern, die genauer dem Original nachgebildet sind, als sie das Steuergerät verarbeiten kann.

Eine weitere Beschränkung des HiL stellt die Tatsache dar, dass üblicherweise alle Tests in Realzeit ablaufen müssen, da das zu testende Steuergerät nicht beliebig beschleunigbar ist. Selbst wenn eine entsprechende Funktion im Steuergerät vorgesehen ist, könnte mit den beschleunigten Tests nicht nachgewiesen werden, dass sich das Steuergerät auch mit normaler Geschwindigkeit korrekt verhält. Damit ist aber direkt die erreichbare Testabdeckung beschränkt. Selbst mit einem verbesserten Prüfstand lässt sich die Anzahl der abgedeckten Testfälle nicht erhöhen. Eine vollständige Abdeckung aller

möglichen Situationen scheidet bei komplexeren Systemen damit praktisch aus.

2.3.3 Rapid Prototyping

Die Grundidee des Rapid Prototyping (RP) bei der Entwicklung von elektronischen Steuergeräten besteht darin, ein simuliertes Modell des zu entwickelnden Steuergerätes schon möglichst frühzeitig im Designprozess testen zu können (siehe 2.5). Damit wird es möglich, schon früh im Designprozess verschiedene Designalternativen unter realen Bedingungen zu evaluieren. So können zum Beispiel verschiedene Algorithmen verglichen werden, ohne das dafür schon eine Hardwareplattform entwickelt werden müsste.

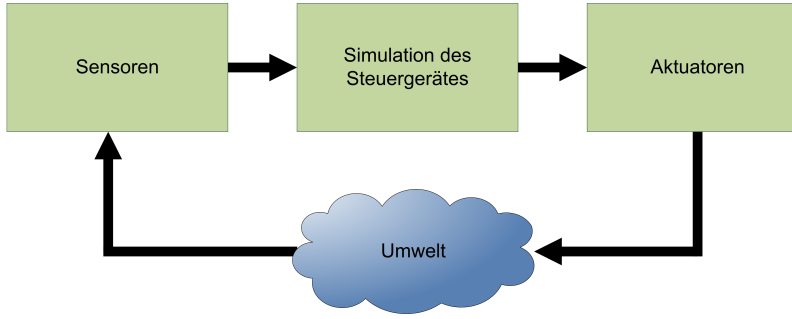


Abbildung 2.5: Prinzip des Rapid Prototyping

Der zentrale Punkt der Rapid Prototyping Umgebung ist ein Simulationsrechner, in dem das Modell des Steuergerätes simuliert werden kann. Die Leistungsfähigkeit dieses Rechners ist dabei ein bestimmender Faktor für die Komplexität des zu simulierenden Steuergerätes. Der Simulationsrechner wird ergänzt um Schnittstellenmodule, die die Verbindung zur realen Umgebung herstellen. Diese beinhalten nicht nur Pegelwandler und Umsetzer zur Anpassung der im Rechner verwendeten Pegel an die der Umgebung, sondern vor allem auch Sensoren und Aktuatoren. Diese nehmen Informationen aus der Umgebung auf (z.B. Drehzahlsensor, Stromsensor, Endschalter) oder wirken auf diese ein (z.B. Motortreiber). Ähnlich wie beim HiL

(siehe Kapitel 2.3.2) stellen diese Sensoren und Aktuatoren eine Einschränkung bei den Einsatzmöglichkeiten für den Einsatz des RP dar. Diese Ein- und Ausgabeschnittstellen sind im Allgemeinen spezifisch für eine Gruppe von Anwendungen. Ein Steuergerät, das Größen benötigt, die der Prüfstand nicht aufnehmen oder erzeugen kann, lässt sich damit nicht simulieren.

Zur Beschreibung der Funktionalität des Steuergerätes stehen verschiedene Methoden zur Verfügung. So kann das gewünschte Verhalten in Form von Blockdiagrammen (z.B. mit Matlab Simulink) dargestellt werden. Ebenso ist eine Beschreibung als SystemC-Modell oder auch mit einer Hardware-Beschreibungssprache (z.B. VHDL) denkbar. Damit lassen sich mit Rapid Prototyping sehr schnell erste Ergebnisse auf Designebene zur Wahl zwischen verschiedenen Alternativen erzielen. Auch im späteren Verlauf der Entwicklung, zum Beispiel bei der Umsetzung in eine Hardwarebeschreibungssprache können damit schon erste Tests vorgenommen werden, ohne dass das Gerät verfügbar ist. Einschränkend für die Komplexität der zu untersuchenden Modelle ist dabei die Forderung nach der Realzeitfähigkeit des Systems (siehe Definition 2.1). Wird das Modell zu komplex für die vorhandene Rechenleistung, erfolgen Reaktionen des Modells nicht mehr rechtzeitig zu den parallel ablaufenden Prozessen. Dies kann zu Fehlern führen und macht die Ergebnisse des Tests unbrauchbar.

Weiterhin ist durch die direkte Anbindung an die reale Umgebung auch die maximal erreichbare Testgeschwindigkeit beschränkt. Selbst wenn die zur Simulation des Steuergerätes verwendeten Rechner über ausreichend Rechenleistung verfügen, ist die Geschwindigkeit doch auf die Geschwindigkeit der Umgebung beschränkt, da diese Prozesse im Allgemeinen nicht beschleunigt werden können. Analog zu HiL ist damit auch beim Rapid Prototyping die erzielbare Testabdeckung beschränkt und lässt in der Praxis keine vollständige Verifikation zu.

Ein weiterer Faktor, der eine vollständige Testabdeckung mittels Rapid Prototyping verhindert, ist der Mangel an Wiederholgenauigkeit für die durchzuführenden Tests. Dies liegt darin begründet, dass die Tests in einer realen Umgebung stattfinden, so dass sich nicht beliebig die immer gleichen Bedingungen einstellen lassen. Während dies für einen einzigen Satz von Parametern noch mit einigem Aufwand realisierbar erscheint, dürfte die exakte Wiederholung von komplexeren Abläufen sehr schwer bis unmöglich sein. Es besteht immer die Gefahr von nicht erkannten Abweichungen in der Umgebung, die zu anderen Ergebnissen beim simulierten Steuergerät führen.

Zudem ist es in der Realität nicht möglich, beliebige Situationen für den Test zu generieren. Im Allgemeinen ist der Simulationsrechner mit allen Zusatzgeräten deutlich größer und schwerer als das später zu realisierende Steuergerät. Ein Einsatz an der dafür vorgesehenen Stelle ist damit schon allein aus Platzgründen nicht möglich. Außerdem gibt es eine Reihe von Testfällen, die eine Beschädigung oder Zerstörung des Testsystems bedeuten könnten und die deswegen nicht beliebig oft herbeigeführt werden können. Damit ist also auch mittels Rapid Prototyping bei komplexeren Systemen keine vollständige Testabdeckung und erst recht keine Garantie für die korrekte Umsetzung einer vorgegebenen Funktionalität gegeben.

2.4 Formale Verifikation

Wie in den vorangegangen Unterkapitel beschrieben, stellen simulative Methoden einen weit verbreiteten Ansatz zur Verifikation von elektronischen Systemen dar. Aufgrund ihrer Charakteristik ist es damit sehr gut möglich, einfache, leicht zu entdeckende Fehler aufzufinden. Allerdings versagen die simulativen Methoden häufig bei den so genannten Corner-Cases (Randfälle), die nur unter ganz speziellen Randbedingungen auftreten. Besonders bei größeren und komplexeren Systemen ist es mit Hilfe der Simulation und externen Stimuli oft nur möglich, einen kleinen Teil des Gesamtsystems anzuregen und damit zu überprüfen. Selten eingenommene Systemzustände werden durch Simulationen auf Grund der beschränkten Simulationszeit teilweise nicht erreicht und damit auch nicht überprüft. Zwar können diese Randfälle durchaus durch gezieltes Anlegen von Stimuli erreicht werden, dazu ist jedoch ein profundes Wissen über das System und das mögliche Auftreten solcher Randfälle nötig. So ergibt es sich, dass in komplexeren Systemen Teile des möglichen Verhaltens nicht überprüft und eventuell vorhandene Fehler unentdeckt bleiben.

Ein weiteres Problem ist die in Kapitel 3.2 beschriebene Problematik bei der Integration von Teilkomponenten in ein Gesamtsystem. Aktuelle Systeme bestehen häufig aus mehreren Teilkomponenten, die später zu einem Gesamtsystem zusammengesetzt werden. Zzum Beispiel ist es bei Automobilherstellern so, dass diese Teilkomponten von Zuliefern beziehen, die wiederum selbst weitere Teile ihres Produktes von anderen Zulieferern einkaufen. Diese Kette kann sich über mehrere Ebenen fortsetzen. Während ein-

zelne Komponenten direkt beim Zulieferer getestet werden, ist der vollständige Test der Funktionalität des Gesamtsystems erst beim Hersteller möglich. Fehler in Teilkomponenten, die erst in der Phase der Entwicklung aufgedeckt werden, können sehr hohe Kosten und Zeitverzögerungen für den Produktionsstart verursachen. Aus diesem Grund ist es wünschenswert, das die Spezifikationskonformität einer Komponente frühzeitig garantiert werden kann. Hier kann die formale Verifikation eine Alternative und Ergänzung zu simulativen Techniken darstellen.

Formale Verifikation verwendet dazu ein mathematisches Modell eines Designs. Aus der Analyse dieses mathematischen Modells kann dann bewiesen werden, dass das System ein bestimmtes Verhalten für alle möglichen Eingangsbelegungen aufweist. Damit ist es prinzipiell möglich, alle Fehler in einem Design zu finden. Auf Grund dieses Vorteils gegenüber simulativen Ansätzen, bekommt die formale Verifikation eine immer stärkere Bedeutung beim Design komplexer elektronischer Systeme. Eine vereinfachte Übersicht der dabei zum Einsatz kommenden Verfahren ist in Abbildung 2.6 zu sehen. Für die formale Verifikation existieren dabei zwei hauptsächliche Bereiche: Die Äquivalenzprüfung (Equivalence Checking) und die Eigenschaftsprüfung (Property Checking).

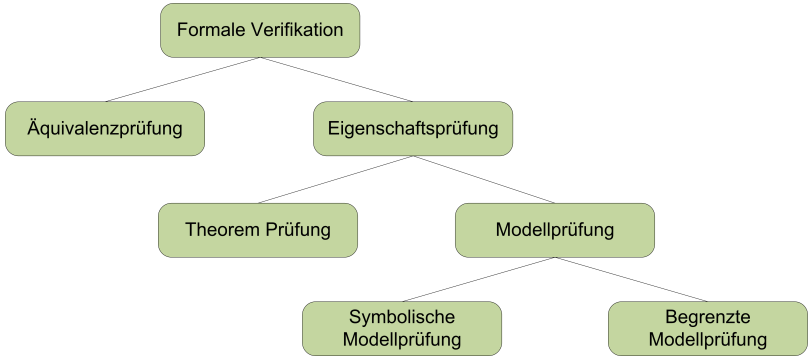


Abbildung 2.6: Verschiedene Verfahren der formalen Verifikation

2.4.1 Äquivalenzprüfung

Bei der Äquivalenzprüfung soll nachgewiesen werden, dass zwei verschiedene Modelle des gleichen Systems funktional äquivalent sind. Zwei verschiedene Modelle sind dann äquivalent, wenn sie für alle möglichen Eingangsbelegungen immer das gleiche Verhalten an den Ausgängen zeigen. Verwendet wird die Äquivalenzprüfung immer dann, wenn Modifikationen oder Optimierungen eines Modells durchgeführt werden. Diese Änderungen können sowohl manuell durch einen Entwickler als auch automatisch zum Beispiel durch ein Synthesetool erfolgen. Sowohl durch menschliche Fehler als auch durch Fehler im Synthesewerkzeug kann es dazu kommen, dass sich das Verhalten des Modells vor und nach der Transformation unterscheidet. Da sich solche Fehler unter Umständen unbemerkt von einem Schritt zu nächsten fortsetzen können, muss nach jedem Schritt nachgewiesen werden, dass die Funktionalität vor und nach der Umwandlung gleich ist. Diese Aufgabe wird mit den Methoden der Äquivalenzprüfung erfüllt.

Ein Nachteil der Äquivalenzprüfung ist jedoch, dass mit ihrer Hilfe nur bewiesen werden kann, dass zwei Modell exakt das selbe Verhalten aufweisen. Es kann damit jedoch nicht nachgewiesen werden, ob dieses Verhalten auch korrekt im Sinne einer Spezifikation ist.

2.4.2 Eigenschaftsprüfung

Im Gegensatz zur Äquivalenzprüfung wird bei der Eigenschaftsprüfung nachgewiesen, dass ein Design auch wirklich das vorgesehene Verhalten abbildet. Normalerweise wird zu Beginn des Designprozesses aus einer eher informellen Beschreibung wie einer Spezifikation die erste Iteration des Designs erzeugt. Dieser Prozess ist nicht zu formalisieren und kann deswegen zu Fehlern bei der manuellen Umsetzung führen. Gleichzeitig lassen sich jedoch aus der Spezifikation auch so genannte Eigenschaftsbeschreibungen oder Operationen erstellen, die das gewünschte Verhalten des Systems wiedergeben. Mit Hilfe der Eigenschaftsprüfung kann dann nachgewiesen werden, dass das Design immer die Eigenschaften erfüllt. Damit ist es schon zu einem relativ frühen Zeitpunkt im Designprozess möglich, eventuelle Fehler bei der Umsetzung der Spezifikation in ein Design aufzudecken und damit Zeit und Kosten im Designprozess zu sparen.

Allerdings muss darauf hingewiesen werden, dass sowohl das Erstellen des Designs aus der Spezifikation als auch das Ableiten der Eigenschaftsbeschreibungen manuelle Prozesse sind, bei denen Fehler auftreten können. So kann die Spezifikation selbst unvollständig oder falsch sein, als auch bei deren Interpretation Fehler gemacht werden. Eine echte Garantie für eine korrekte Umsetzung der Spezifikation in ein Design kann also auch die Eigenschaftsprüfung nicht liefern. Jedoch ermöglicht es die doppelte Interpretation der Spezifikation durch den Entwickler und durch den Verifikateuer, dass eventuelle Fehler beim Vergleich der beiden Darstellung gefunden werden.

Um zu überprüfen, ob ein Design die aus der Spezifikation abgelesenen Eigenschaften erfüllt, gibt es zwei grundsätzliche Herangehensweisen. Beim Theorembeweis handelt es sich um einen manuellen Vorgang, bei dem Beweise von einem Menschen geführt werden. Der Computer hat hier nur unterstützende Funktion. Diese Methode erfordert großes Fachwissen und Erfahrung und kann deswegen im Allgemeinen nur von Experten angewendet werden. Auf der anderen Seite steht die Modellprüfung (Model Checking), bei dem es sich um einen vollautomatischen Vorgang handelt. Eine Interaktion ist dabei nur bei Erstellen der Eigenschaftsbeschreibung erforderlich.

Im Rahmen dieser Arbeit kommt nur die Modellprüfung zum Einsatz, weswegen hierauf nun etwas genauer eingegangen werden soll.

2.4.2.1 Modellprüfung (Model Checking)

Für die Modellprüfung ist es erforderlich, sowohl das Design als auch die Beschreibung der Eigenschaften in ein mathematisches Modell zu überführen.

Dabei wird das mathematische Modell des Designs dazu verwendet, das kombinatorische und sequentielle Verhalten des Designs zu erfassen. Unter dem kombinatorischen Verhalten ist der Zusammenhang zwischen den Eingängen und den Ausgängen des Designs zu einem festen Zeitpunkt gemeint. Zur Darstellung dieses Verhaltens wird bei digitalen Schaltungen häufig die Boolesche Algebra verwendet. Das sequentielle Verhalten eines Designs hingegen beschreibt den Zusammenhang zwischen den Ein- und Ausgängen über einen zeitlichen Verlauf. Zu dessen Beschreibung können Finite State Machines (FSMs) oder Zustandsübergangsdiagramme (State Transition Graph (STG)) verwendet werden.

Für die Beschreibung des gewünschten Verhaltens eines Designs werden so genannte Operationen definiert. Diese Operationen beschreiben das Verhalten eines Schaltung über einen kürzeren Zeitraum und macht sie so für die Verifikation besser handhabbar. Die Gesamtfunktionalität eines Designs setzt sich also aus einer Vielzahl einzelner Operationen zusammen. Diese Operationen oder Eigenschaften einer Schaltung können auch als Automaten dargestellt werden. Viele dieser Operationen lassen sich dabei als vereinfachte Timingdiagramme darstellen, wie sie zum Beispiel in [4] vorgestellt werden. Diese Diagramme lassen sich auch in der von der Firma OneSpin [18] für ihre Toolkette verwendeten Sprache InTerval Language (ITL) umsetzen und sie so mit Computern nutzbar machen. Diese Toolkette wurde in dieser Arbeit verwendet. Die so erstellten temporalen Eigenschaften werden Operationseigenschaften genannt.

Die Aufgabe der Modellprüfung ist es nun, die Gültigkeit dieser Operationseigenschaften für das mathematische Modell des Designs nachzuweisen. Dazu kann das Verhalten des Designs als Graph dargestellt werden, der aus allen möglichen Zuständen des Systems besteht. Die Einhaltung der Operationseigenschaften muss dann für den vollständigen Graphen überprüft werden. Es ist offensichtlich, dass der Umfang des zu untersuchenden Graphen und damit die Laufzeit des Algorithmus exponentiell mit der Anzahl an Zuständen im Graphen ansteigt. Dieses Problem ist als „State Explosion“ bekannt und begrenzt die Größe eines Designs, dass in der Praxis noch sinnvoll untersucht werden kann. Die Möglichkeiten zur formalen Verifikation hängen also direkt von der Qualität der zu Grunde liegenden Datenstrukturen und mathematische Modell ab.

Eine Möglichkeit zur effizienten Darstellung von Booleschen Funktionen sind Binary Decission Diagrams (BDDs), die von Bryant [6] vorgeschlagen wurden. Fast jede Funktion, die von praktischer Bedeutung ist, lässt sich mit Hilfe eines BDD darstellen. Außerdem ist es so, dass jeder Funktion exakt ein BDD zugeordnet ist. Die entscheidende Eigenschaft ist jedoch, dass die Komplexität zur Verarbeitung von BDDs eine polynomiale Abhängigkeit von der Größe der BDDs hat. Sie sind damit besonders gut für die Anwendung bei der formalen Verifikation geeignet.

Ein weiterer Ansatz zur Verbesserung der Anwendbarkeit der formalen Verifikation sind effiziente Lösungen für das sogenannte Erfüllbarkeitsproblem (SAT, von englisch *satisfiability*). Unter einem SAT-Problem versteht man normalerweise die konjunktive Normalform einer Booleschen Funktion. Die Auf-

gabe eines SAT-Lösers ist es, eine Belegung der Variablen der gegebenen Funktion zu finden, so dass das Ergebnis der Funktion „wahr“ wird. Hier wurden in den letzten Jahren immer effizientere Verfahren entwickelt, die auch große Probleme mit mehreren Tausend Variablen lösen können.

Beim so genannten *Bounded Model Checking (BMC)* wird zur Reduktion der Komplexität die Schaltung und die zu erfüllenden Eigenschaften für eine begrenzte Anzahl k von Schritten ausgerollt. Das Ausrollen beginnt dabei immer vom Startzustand (nach einem Reset). Sodann wird die Einhaltung der Eigenschaft für dieses Zeitfenster überprüft. Wird dabei kein Gegenbeispiel gefunden, so wird die Anzahl k der Schritte im zu untersuchenden Zeitfenster erhöht und ein neuer Durchlauf gestartet. Dieses Verfahren wird so lange fortgesetzt, bis entweder ein Gegenbeispiel gefunden wurde oder aber die Anzahl k der zu untersuchenden Schritte eine vorgegebene Grenze überschritten hat. Mit Hilfe von BMC ist also nur der Nachweis möglich, dass eine Eigenschaft nicht erfüllt wird oder dass sie zumindest für das untersuchte Intervall erfüllt wurde. Eine allgemeine Aussage, ob die Eigenschaft für alle möglichen Abfolgen erfüllt werden kann, ist damit nicht möglich.

Ein ähnlicher Ansatz kommt beim *Interval Property Checking (IPC)* zum Einsatz. Auch hier wird das Verhalten der Schaltung und die zu überprüfende Eigenschaft auf ein endliches Zeitfenster ausgerollt. Auch hier wird das Problem wieder auf ein SAT-Problem zurück geführt, dass mit entsprechenden Verfahren bearbeitet wird. Im Unterschied zum BMC beginnt das Abrollen und Überprüfen der Eigenschaft jedoch nicht immer im Reset-Zustand, sondern kann jederzeit erfolgen. Die Eigenschaften sind deswegen so formuliert, dass sie sich zum einen nur auf ein begrenztes Zeitfenster beziehen, aber gleichzeitig nicht auf absolute Zeitpunkte festgelegt sind. Mit Hilfe dieses Ansatz ist es nicht mehr notwendig, das zu untersuchende Zeitfenster in jedem Schritt weiter zu vergrößern. Findet der SAT-Solver also kein Gegenbeispiel, so ist nachgewiesen, dass die Eigenschaft für alle möglichen Startbedingungen erfüllt ist. Das Auftreten eines Gegenbeispiels kann nun jedoch verschiedene Bedeutungen haben. Dazu soll zuerst kurz auf die Klassifikation von Fehlern eingegangen werden.

2.4.3 Erreichbarkeit

Bei der Beurteilung, ob ein zu untersuchendes System die Vorgaben der Spezifikation erfüllt, gibt es grundsätzlich nur zwei Möglichkeiten. Entweder

die Spezifikation wird vom System erfüllt oder sie wird nicht erfüllt. Ebenso liefert der Test durch das Verifikationsverfahren nur zwei Ergebnisse. Er kann positiv (die Spezifikation wird erfüllt) oder negativ (die Spezifikation wird nicht erfüllt) sein. Aus der Kombination von den tatsächlichen Eigenschaften des Systems und der Entscheidung des Eigenschaftsprüfers ergeben sich damit vier mögliche Fälle, die in einer Wahrheitsmatix (Tabelle 2.1) dargestellt werden können.

	Spezifikation erfüllt	**Spezifikation nicht erfüllt**
Test positiv	richtig positiv (TP)	falsch positiv (FP)
Test negativ	falsch negativ (FN)	richtig negativ (TN)

Tabelle 2.1: Wahrheitsmatrix mit zwei Dimensionen

1. **Richtig positiv**: Das System erfüllt die Spezifikation und der Test hat dies richtig angezeigt.
2. **Falsch negativ**: Das System erfüllt die Spezifikation, der Test hat es aber als fehlerhaft eingestuft.
3. **Falsch positiv**: Das System erfüllt die Spezifikation nicht, wurde aber vom Test als korrekt eingestuft.
4. **Richtig negativ**: Das System erfüllt die Spezifikation nicht und der Test hat dies richtig angezeigt.

Nur für die Fälle 1 (richtig positiv) und 4 (richtig negativ) entspricht also das Ergebnis des Tests der Wirklichkeit, der Test hat richtig gearbeitet. Im ersten Fall ist das System fehlerfrei und wird auch so bewertet, während im vierten Fall ein Fehler im System vorhanden ist, für den der Eigenschaftsprüfer ein Gegenbeispiel liefert. Im Fall 3 (falsch positiv) hingegen trat ein Fehler im System auf, der nicht erkannt wurde. Unter der Annahme, dass der Eigenschaftsprüfer selbst fehlerfrei arbeitet, bedeutet das, dass die Spezifikation nicht korrekt in eine Eigenschaftsbeschreibung überführt wurde. Diese Problematik und deren mögliche Behebung liegt jedoch nicht im Fokus dieser Arbeit.

In Fall 2 (falsch negativ) erfüllt das zu untersuchende System zwar die Spezifikation, jedoch wird vom Eigenschaftsprüfer ein Gegenbeispiel geliefert, das System also als nicht der Spezifikation entsprechend bewertet. Dies ist

normalerweise dann der Fall, wenn die Eigenschaftsbeschreibungen zu allgemein gehalten sind. Der Eigenschaftsprüfer liefert dann ein Gegenbeispiel, dass so im realen Betrieb nie auftreten kann. Der Grund dafür liegt darin, dass nicht alle theoretisch möglichen Zustände des Systems im realen Betrieb eingenommen werden können.

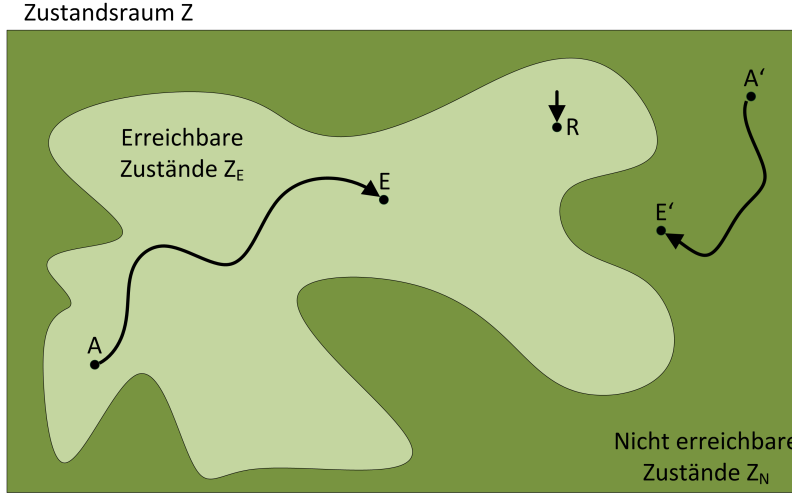


Abbildung 2.7: Erreichbarkeit von Zuständen

Dieses Problem der Erreichbarkeit soll anhand von Abbildung 2.7 näher erläutert werden. Aus der Kombination aller Zustandsgrößen des Systems ergibt sich der gesamte Zustandsraum Z aller möglichen Zustände. Im realen Betrieb wird dieser vollständige Zustandsraum Z jedoch dadurch eingeschränkt, dass das System immer von einem exakt definierten Punkt startet. Dies ist der Zustand R, der direkt nach einem Reset des Systems eingenommen wird. Ausgehend von diesem Startzustand können dann unter Umständen nur noch eine begrenzte Menge $Z_E \subseteq Z$ von Zuständen erreicht werden. Gültige Pfade für die Verifikation starten innerhalb der Menge der erreichbaren Zustände Z_E und enden auch dort. Für ein falsch negatives Ergebnis würde nun der Eigenschaftsprüfer einen Startzustand A' aus der Menge der im realen Betrieb nicht erreichbaren Zustände Z_N auswählen und von dort

einen Pfad finden, der nicht zum zu beweisenden Endzustand *E* führt. Für dieses Gegenbeispiel ist zwar die Eigenschaftsbeschreibung nicht erfüllbar, jedoch wird dieser Fall im realen Betrieb nie auftreten.

Ein Beispiel dafür ist der fehlende Ausschluss des Reset-Eingangs bei einer "´normalen“ Eigenschaftsbeschreibung. Der Eigenschaftsprüfer kann dann zur Laufzeit der Eigenschaft einen Reset annehmen, so dass der gewünschte Endzustand nicht erreicht wird. Weiterhin könnten Zustände im System existieren (z.B. eine Kombination von Zuständen verschiedener FSMs), die zwar theoretisch möglich sind, in einem fehlerfreien Betrieb in dieser Kombination nicht auftreten können. Ohne weitere Einschränkungen bei den Vorgaben würde der Eigenschaftsprüfer hier ein Gegenbeispiel generieren, indem er einen der nicht-erlaubten Zustände als Ausgangspunkt nimmt.

Um diese „False-Negatives“ auszuschließen, müssen also die unerreichbaren Anfangszustände durch das Hinzufügen von zusätzlichen Erreichbarkeitsbedingungen ausgeschlossen werden. Diese Bedingungen können teilweise automatisch ermittelt werden. Allerdings ist das nicht für alle Fälle möglich, so dass hier das manuelle Erweitern der Erreichbarkeitsbedingungen durch den Verifkateur erforderlich ist. Dieser verfügt zum Beispiel über das nötige Verständnis für die Schaltung oder bekommt dies aus Gesprächen mit dem Entwickler, um so die passenden Erreichbarkeitsbedingungen zu erstellen und damit „false-negatives“ auszuschließen.

Das Verfahren des IPC ermöglicht damit den effizienten Beweis von Eigenschaften bei gleichzeitig handhabbarer Laufzeit für den Nachweis durch die begrenzte Größe des zu untersuchenden Zeitfensters. Allerdings ist auch bei diesem Verfahren die Laufzeit noch abhängig von der Größe des Fensters und der Anzahl der betrachteten Größen, so dass keine beliebig großen Probleme in sinnvoller Zeit lösbar sind. Das IPC geht auf Entwicklungen der Firma Siemens zurück. Es wird heute als kommerzielles Produkt von der Firma OneSpin Solutions [18] vertrieben. Für die Umsetzung der in dieser Arbeit vorgestellten Methodik kamen die Werkzeuge von OneSpin Solutions zum Einsatz. Aus diesem Grunde soll nun im Folgenden etwas detaillierter darauf eingegangen werden.

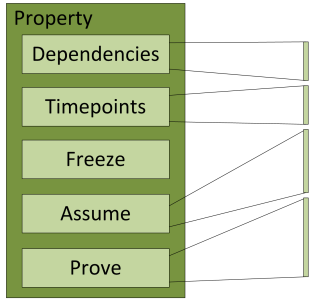
2.5 OneSpin 360® MV

2.5.1 Eigenschaftsbeschreibungen (Properties)

Für die Verifikation von Schaltungen in OneSpin 360® MV werden zwei Komponenten benötigt. Zum einen die Beschreibung der Schaltung selbst. Diese kann in Form einer Hardwarebeschreibungssprache wie VHDL oder Verilog in das Tool eingelesen werden. Im Allgemeinen werden diese Daten bei der Verifikation vom Entwickler bereit gestellt und vom Verifikateur nicht verändert. Eine eventuell notwendige Korrektur von Fehlern wird vom Entwickler vorgenommen, der daraufhin den überarbeiteten Quellcode dem Verifikateur wieder zur Verfügung stellt.

Die zweite benötigte Komponente ist die Beschreibung des nachzuweisenden Systemverhaltens. Diese Eigenschaftsbeschreibungen werden in OneSpin 360® MV als *Properties* bezeichnet. Jede Property beschreibt das gewünschte Systemverhalten für eine kurzes Zeitintervall. Dazu enthält die Property sowohl Annahmen zum aktuellen Zustand des Systems, zu den Eingangsbelegungen, die eine Reaktion des Systems bewirken sollen und schließlich die Festlegung des zu erwartenden korrekten Systemverhaltens auf diese Eingänge. Zur Beschreibung der Properties können sowohl die proprietäre Sprache InTerval Language (ITL) als auch System Verilog Assertions (SVA) eingesetzt werden. Die Syntax der in dieser Arbeit verwendeten ITL ist bekannten Hardwarebeschreibungssprachen ähnlich, wurde jedoch um die Möglichkeit erweitert, zeitliche Bedingungen zu beschreiben. So existieren hier neben Zeitvariablen auch zeitliche Operatoren zum Beispiel für Wann-Dann-Beziehungen. Zur Beschreibung des Zeitverhaltens wird nicht mit absoluten Zeiten, sondern mit abstrahierten, relativen Zeiten gearbeitet. Die kleinste zu betrachtende Zeiteinheit entspricht dabei einem Taktschritt des Systems, unabhängig von dessen tatsächlicher Taktrate. Dies macht es unter Umständen erforderlich, im Spezifikationen vorgegebene absolute Zeitwerte in die relativen Werte für die Eigenschaftsbeschreibungen umzusetzen.

In Abbildung 2.8 ist der Quellcode einer einfachen Beispielproperty zu sehen. Der Quellcode wurde aus der OneSpin Benutzerdokumentation [8] entnommen.




```
1 property grant_master_1 is
2 dependencies:
3     no_reset;
4 for timepoints:
5     t_free_i = t+4;
6 assume:
7     at t: state_s = IDLE;
8     at t: request_i(1 downto 0) = "10";
9 prove:
10    at t+1: grant_o = "010";
11    during [t+2,t_free_i+1]: grant_o = 0;
12    at t_free_i+1: state_s = IDLE;
13 end property;
```

Abbildung 2.8: Beispiel-Property (aus [8])

Die Properties können jeweils in getrennten Dateien abgelegt werden. Es ist jedoch auch möglich, mehrere Properties in einer einzigen Datei zusammen zu fassen. Der Anfang einer neuen Property wird durh das Schlüsselwort *property* gefolgt von dem Namen der Property gekennzeichnet. Der Name der Property muss dabei einmalig sein. Geschlossenen wird die Property mit den Schlüsselwörtern *end property*.

Dependencies: Constraints und Assertions Den nächsten Abschnitt einer Property bildet die so genannte *Dependency-List*. Hier werden Annahmen getroffen, die als Ausgangsbasis für den Beweis der Eigenschaft angenommen werden. Dies können sowohl so genannte Constraints als auch Assertions sein. Beiden gemeinsam ist, dass sie die Menge der möglichen Zustände einschränken, indem zum Beispiel einzelne Startwerte festgelegt und/ oder andere Werte ausgeschlossen werden. Die Dependencies sind unter anderem auch notwendig, um "´False Negatives“ auszuschließen (siehe Abschnitt 2.4.3).

Unter Constraints versteht man bei OneSpin 360® MV Vorgaben, die ohne weiteren Beweis als richtig angenommen werden. Sie sollten sich deswegen nach Möglichkeit nur auf Eingangssignale beziehen, die ja vom System nicht beeinflusst werden können. Dies können Vorgaben sein, die direkt aus der Spezifikation entnommen werden. Ein Beispiel ist der Ausschluss von bestimmten Kombinationen von Eingangssignalen, die laut Spezifikati-

on im korrekten Betrieb nicht auftreten dürfen. Ein weiteres Beispiel für die Verwendung von Constraints ist die Bedingung, dass während der gesamten Laufzeit einer Property das Auftreten eines Reset-Signals ausgeschlossen wird. Die korrekte Behandlung des Reset-Signals wird in einer eigenen Property beschreiben, sollte dann aber bei den Properties für den „normalen" Betrieb ausgeschlossen werden. In der Beispiel-Property (Abbildung 2.8) erfolgt dies in Zeile 3 mit Hilfe des Makros `no_reset`. Ansonsten stellt das Auftreten eines Resets ein klassisches Gegenbeispiel dar, warum eine Property nicht erfüllt wird. Mit Hilfe der Constraints kann somit das Verhalten der Umgebung festgelegt werden, wie es sich aus der Spezifikation oder durch andere, schon verifizierte Komponenten ergibt. In den Constraints sollten alle Bedingungen erfasst werden, die sich durch das zu verifizierende System nicht beeinflussen lassen.

Das Setzen dieser Constraints stellt einen kritischen Teil bei der Verifikation dar. Zum einen ist es häufig notwendig, den Suchraum für die formale Verifikation einzuschränken, um *False-Negatives* zu verhindern. Auch ermöglichen die Contraints die Abbildung des Verhaltens der Umgebung, und schränken damit die Verifikation auf sinnvolle Eingangskombinationen ein. Auf der anderen Seite ist es jedoch auch möglich, das durch die Verwendung von zu vielen oder auch falschen Constraints im System vorhandene Fehler nicht entdeckt werden, da der Fall, der zu einem Auftreten des Fehlers führt bei der Verifikation ausgeschlossen wird. Wird also bei der Verifikation nur von einem korrekten Verhalten der Umgebung ausgegangen und dies durch die passenden Constraints abgebildet, so kann damit die Reaktion des zu verifizierenden Systems auf ein externes Fehlerverhalten nicht überprüft werden. Hier obliegt die Verantwortung für das korrekte Setzen von Constraints alleine dem Verifikateur, da die Constraints vom Verifikationswerkzeug ungeprüft als korrekt übernommen werden.

Eine weitere Möglichkeit, Vorgaben für das Verhalten des Systems zu machen stellen die *Assertions* dar. Auch diese Vorgaben werden, ähnlich wie bei den Constraints, innerhalb einer Property als gegeben und korrekt angenom-

men. Im Gegensatz zu Constraints, die Annahmen zu den Eingangssignalen machen, treffen Assertions jedoch Annahmen zu dem zu testenden System selbst. Aus diesem Grund können und müssen diese Annahmen auch einmal während des Verifikationsprozesses nachgewiesen werden. Constraints hingegen werden nicht beweisen. Ein Beispiel für die Verwendung von Assertions sind Aussagen über zulässige Zählerstände oder Kombinationen von Zuständen der im System vorhandenen FSMs. So kann damit an zentraler Stelle nachgewiesen werden, dass ein Zähler unter keinen Umständen einen bestimmten Wert erreichen kann. Diese Annahme kann danach in weitere Properties übernommen werden, und verhindert dadurch das Auftreten von *False-Negatives*, ohne dass diese jedes mal explizit ausgeschlossenen werden müssen.

Die Verlässlichkeit bei der Verwendung von Assertions ist also deutlich höher, da diese im Gegensatz zu Constraints immer auch bewiesen werden müssen. Allerdings sind Assertions auch wieder auf das System selbst beschränkt und können zur Beschreibung des Umgebungsverhaltens nicht eingesetzt werden.

Time Points Die Beschreibung der zu überprüfenden Eigenschaften innerhalb einer Property hat immer auch einen Zeitbezug. Oft wird dabei nicht nur ein einziger Zeitpunkt betrachtet, sondern das nachzuweisende Verhalten erstreckt sich über eine Zeitspanne. Gerade bei der Spezifikation von Protokollen kommen immer wieder fest vorgegebene Zeitspannen zum Einsatz, innerhalb derer eine Reaktion des Systems erfolgen soll. Um nun dieses Verhalten in einer Property darzustellen, kann prinzipiell mit festen Zeitwerten gearbeitet werden. Im Sinne einer Flexibilisierung und zur Verbesserung der Lesbarkeit bietet es sich jedoch an, diese Zeitwerte mit einem Symbolischen Namen zu versehen. So wird im Bespiellisting (Abbildung 2.8) mit `t_free_i = t +4` der Zeitpunkt `t_free_i` definiert, der vier Zeiteinheiten nach Beginn der Property liegt. Dieser Name kann nun im weiteren Verlauf der Property verwendet werden. Sollte eine Änderung des Wertes notwendig sein, so

kann diese an zentraler Stelle im Kopf der Property vorgenommen werden. Alle darauf aufbauenden Werte werden automatisch geändert.

Freeze Ähnlich dem Abschnitt *Time Points*, mit dessen Hilfe Zeitpunkte mit sprechenden Namen versehen werden können, ist das über den Abschnitt *Freeze* auch mit Signalen möglich. Auf diese Art und Weise können neue Konstanten definiert werden, die den Wert eines Signals zum angegebenen Zeitpunkt enthalten. Selbst wenn sich der Wert des Signals später ändert, bleibt er in der Freeze-Konstante unverändert.

Neben der übersichtlichen Darstellung von innerhalb eines Beweises benötigten Signalen lassen sich damit auch sehr einfach Werte zu bestimmten Zeitpunkten festhalten. Auch diese Funktionalität kann bei der Protokollverifikation genutzt werden, um das zeitliche Verhalten des Systems zu beschreiben.

Assume und Prove Der Abschnitt *Assume* bildet zusammen mit dem Teil *Prove* den eigentlichen Kern des Beweises der Eigenschaft. Im *Assume-Teil* werden die Startbedingungen definiert, die das zu beweisende Verhalten des Systems auslösen sollen. Diese beinhalten neben Eingangssignalen auch interne Zustände und Registerwerte des Systems. Die Werte dieser Parameter können dabei auch zu unterschiedlichen Zeitpunkten bestimmt werden, um so zum Beispiele eine auslösende Folge von Signalen zu beschreiben. Ähnlich zu den Ausführungen über die Constraints gilt auch hier, dass so wenige Annahmen wie möglich getroffen werden sollten. Alle Parameter, deren Wert im Assume-Teil nicht festgelegt wurden, können vom Eigenschaftsprüfer frei belegt werden, um so eventuelle Fehler aufzudecken. Jedes Festlegen von Parametern schränkt den Suchraum hingegen ein. In der Beispiel-Property wird dazu angenommen, dass sich das System zum Zeitpunkt `t` im Zustand `state_s = IDLE` befindet und das Signal `request_i` den Wert `10` hat.

Abschließend erfolgt im *Prove-Teil* das nachzuweisende Verhalten des Systems. Auch hier können analog zum Assume-Teil wieder beliebige Kombinationen von Signalen, Zuständen und Registerwerten zu unterschiedlichen Zeitpunkten als Sollvorgaben festgelegt werden. Im Beispiellisting soll an dieser Stelle nachgewiesen werden, dass zum Zeitpunkt `t+1` das Signal `grant_o` den Wert `010` hat. Ebenso soll nachgewiesen werden, dass das Signal `grant_o` im Zeitraum `t+2` bis `t_free_i+1` konstant den Wert `0` besitzt.

Selbstverständlich können hier jedoch keine Vorgaben über die Eingangssignale gemacht werden. Der Eigenschaftsprüfer überprüft dann, ob die geforderten Endbedingungen unter Annahme der Startbedingungen immer erreicht werden. Dazu werden auch die im Dependencies-Teil angegebenen Randbedingungen berücksichtigt. Für den Fall, dass die Endbedingungen immer erreicht werden, hält die Property und die Eigenschaft ist bewiesen. Ansonsten liefert der Eigenschaftsprüfer ein Gegenbeispiel mit einer Eingangsbelegung und einem Signalverlauf, durch den die Endbedingungen nicht erfüllt werden. Anhand des Gegenbeispiels muss dann entschieden werden, ob es sich tatsächlich um einen Fehler im zu untersuchenden System handelt oder ob ein Fehler bei der Erstellung der Property aufgetreten ist. Dies könnte im Falle eines „False-Negative“ ein unzureichendes Constraining sein, mit dem dieser Fall für den Eigenschaftsbeweis ausgeschlossen werden muss, wenn er in der Realität nie auftreten kann. Fehler können aber auch in den Properties selbst liegen. So können Bedingungen aus der Spezifikation falsch interpretiert worden oder sogar die Spezifikation selbst fehlerhaft sein.

2.5.2 Completeness

Mit Hilfe der im vorherigen Abschnitt beschriebenen Properties werden normalerweise immer nur einzelne Aspekte des gesamten Verhaltens eines System beschrieben. Eine Beschreibung des vollständigen Systemverhaltens in einer einzigen Property ist kaum sinnvoll darstellbar. Für eine vollständi-

ge Beschreibung sind deswegen mehrere Properties nötig. Dabei beschreibt idealerweise jede Property einzelne Teilaspekte des Systemverhaltens und ergänzt sich mit den anderen Properties so, dass das vollständige Verhalten überprüft wird (siehe Abbildung 2.9).

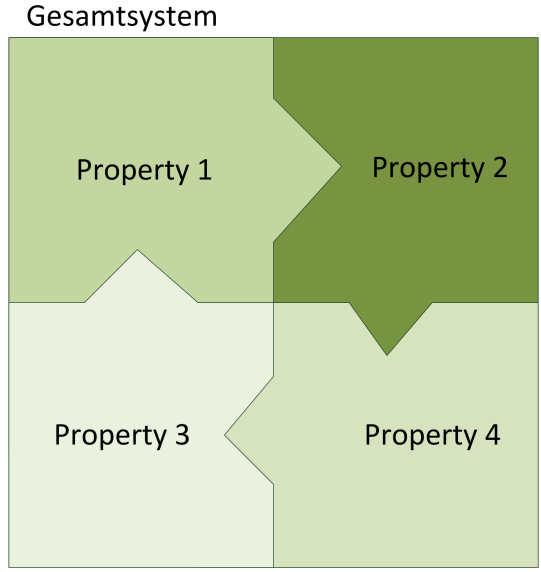


Abbildung 2.9: Prinzip der Completeness

Rein manuell ist es bei komplexeren Systemen sehr schwer zu überblicken, ob man mit den vorhandenen Properties schon das gesamte Verhalten abgedeckt hat. Es müssen dazu sämtliche Kombinationen von Eingangsvariablen mit den internen Zuständen erfasst werden. Außerdem muss auch sichergestellt werden, dass die Ausgänge eines Systems immer feste Werte aufweisen, wenn dies in der Spezifikation so verlangt ist. Zur Lösung dieses Problem wurde von Bormann in seiner Dissertation [4] eine Methode vorgestellt, die es ermöglicht den Nachweis zu führen, dass das vollständige

Verhalten eines Systems durch Properties beschrieben wird. Diese Methodik wird im Tool OneSpin 360® MV eingesetzt.

Der so genannte *Completeness Checker* benötigt dazu weitere Informationen über das System, mit deren Hilfe er die Vollständigkeit der Properties unter den gegebenen Randbedingungen nachweisen kann. Diese Informationen werden in der *Completeness Description* an zentraler Stelle zusammen gefasst. Sie enthält unter anderem Informationen darüber, wie die einzelnen Properties zusammen hängen, welche Ein- und Ausgänge des Systems in die Überprüfung mit aufgenommen werden sollen, sowie weitere Festlegungen für die Vollständigkeitsprüfung. Ein Beispiel für eine solche Completeness Description ist in Listing 2.1 zu sehen. Anhand dieses Beispiels sollen nun kurz die wesentlichen Abschnitte der Beschreibung erklärt werden.

```
1  completeness arbiter is
2    inputs;
3      reset, free_i, request_i;
4    determination requirements:
5      determined(grant_o);
6    reset_property:
7      reset;
8    property_graph:
9      reset, no_request, grant_master_0, grant_master_1, grant_master_2
           ->
10       no_request, grant_master_0, grant_master_1, grant_master_2;
11 end completeness;
```

Listing 2.1: Beispiel einer Completeness Description (aus [8])

Festlegung der Ein- und Ausgänge In der mit dem Schlüsselwort `inputs:` eingeleiteten Liste werden alle Eingänge des zu verifizierenden Systems definiert. Diese Informationen können nicht automatisch aus dem in das Tool eingelesenen Quellcode des Systems entnommen werden, da es mit dem Tool auch möglich ist, nur Teile eines Gesamtsystems zu verifizieren. Das zu verifizierende Teilsystem wird in diesem Zusammenhang als *Cluster* bezeichnet und könnte auch nur ein Teilmodul eines größeren Systems darstel-

len. Ein Beispiel dafür wäre die Intellectual Property (IP) eines Communication Controllers, die Bestandteil einer integrierten Schaltung ist. Die Input-Liste enthält für diesen Fall also alle Signale, die als von außen kommend angesehen werden sollen. Das bedeutet unter anderem, dass ihr Verhalten nicht durch Properties nachgewiesen wird, sondern als vorgegeben angenommen wird.

Analog dazu wird die Liste der Ausgänge des Clusters durch das Schlüsselwort `determination_requirements` bestimmt. Alle hier definierten Signale werden als Ausgänge des Clusters angesehen. Eine wichtige Eigenschaft der Ausgangssignale, die während der Verifikation nachgewiesen werden soll, ist die Determinierung der Ausgänge. Darunter versteht man, dass der Wert der Ausgangssignale bei jeder beliebigen Eingangsbelegung in Kombination mit einem beliebigen internen Zustand immer exakt bestimmbar sein muss. Nur so kann das Verhalten des Systems jederzeit bestimmt werden. Häufig ist es jedoch so, dass die Spezifikationen den Wert der Ausgangssignale nur zu bestimmten Zeitpunkten festlegen und ansonsten beliebige Werte für die Ausgänge erlauben. Um diese Freiheitsgrade auch bei der Verifikation abbilden zu können, erlaubt OneSpin 360® MV auch so genannte *bedingte Determinierungen*. Hiermit ist es möglich anzugeben, dass der Wert eines Ausgangs nur unter bestimmten Bedingungen festgelegt sein muss. Ein Beispiel dafür wäre, dass der Wert eines Ausgangsregisters `register1_out` nur dann determiniert sein muss, wenn das Signal `valid` den Wert `1` hat. Die Syntax für diese Bedingung lautet: `If valid = 1 then determined(register1_out) end if;` Allerdings ist in diesem Fall zu beachten, dass nun das Signal `valid` immer determiniert sein muss.

Über die Liste der Ein- und Ausgänge lässt sich also entweder ein vollständiges System oder auch nur Teile davon für die Verifikation festlegen. Wichtig ist in diesem Zusammenhang jedoch, dass Signale in der Ausgangs-Liste nicht gleichzeitig bei den Eingängen aufgelistet sein dürfen, da die Eingänge explizit von der Überprüfung der Determinierung ausgeschlossen sind.

Property Graph Nach dem Schlüsselwort `property_graph` folgen essentielle Informationen zum Zusammenspiel der einzelnen Properties. Jede Property wird dazu eine oder auch mehrere Nachfolge-Properties zugeordnet. Es entsteht damit eine Kette aus Properties, die zusammen das Verhalten des zu verifizierenden Systems beschreiben. Ausgangspunkt dieser Ketten ist immer die Reset-Property, die das Verhalten nach einem Reset bestimmt. Daran schließen sich dann die weiteren Properties an. Dadurch, dass eine Property mehrere Nachfolger haben kann, ist es natürlich auch möglich, Fallunterscheidungen abzubilden. Ein Beispiel für einen solchen Propertygraphen, wie er sich aus dem in Listing 2.1 ergibt, ist in Abbildung 2.10 zu sehen.

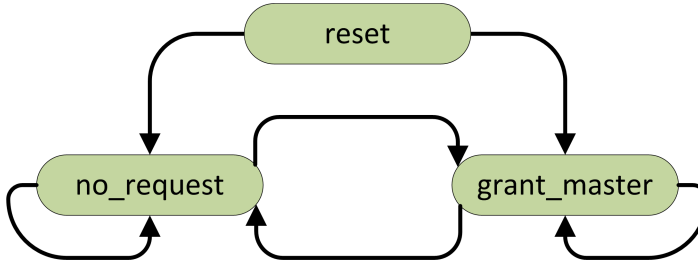


Abbildung 2.10: Beispiel für einen Property Graph

Um diese Ketten zu bilden, ist es erforderlich, dass die einzelnen Properties an ihren Enden zusammen passen müssen. Das bedeutet, dass die Annahmen, die in einer Property getroffen werden, in der vorangegangenen Property bewiesen werden müssen. Nur so lässt sich eine lückenlose Kette von Eigenschaften aufbauen. Wie in Abbildung 2.11 dargestellt ist, gibt es dabei mehrere Bedingungen, um zwei Properties aneinander passen zu lassen. Wie schon im Abschnitt 2.5.1 dargestellt, werden zu Beginn einer jeden Property Annahmen zum Zustand des Systems und zu den Werten der Eingangssignale gemacht. Ebenso werden am Ende einer Property im Prove-Teil sowohl der Zustand des Systems als auch die Ausgangssignale festgelegt.

Damit nun eine Property exakt auf eine weitere Property folgen kann, ist es erforderlich, dass der Ausgangszustand des Systems, der im Assume-Teil der Nachfolgerproperty B angenommen wird, im Prove-Teil der Vorgängerproperty A bewiesen wurde. Sie wirken so wie zwei Puzzle-Teile, die an den Seiten exakt zusammen passen müssen. Es handelt sich hier um die in Abbildung 2.11 angegebenen „Assumptions aus Historie“.

Mit dem bisher beschriebenen Vorgehen wäre es jedoch nur möglich, rein lineare Ketten von Properties zu erstellen, bei denen immer exakt eine Property der vorherigen folgt. Um Fallunterscheidungen vornehmen zu können, sind damit weitere Informationen nötig. So können im Kopf einer Property neben den Assumptions, die direkt von der Vorgängerproperty abhängen noch weitere, davon unabhängige Annahmen getroffen werden. Bei diesen Annahmen handelt es sich üblicherweise um einzelne Werte oder auch zeitliche Verläufe der Eingangssignale des Systems. In Abhängigkeit von diesen Eingängen soll das System ein anderes Verhalten zeigen. Für jede mögliche Eingangskombination wird also eine neue Property mit dem gleichen Historiensatz der Assumptions aber einem anderen Satz von weiteren Eingängen erstellt. Diese passt damit direkt an die Vorgängerproperty, verzweigt dann aber auf einen neuen Pfad. Im Rahmen der Completeness ist es dabei von Bedeutung, dass die einzelnen Properties jeweils sauber an die Vorgängerproperty anschließen, und andererseits auch alle möglichen Kombinationen der Eingangssignale abgedeckt sind. Das bedeutet, dass für jede mögliche Eingangskombination eine Folgeproperty existieren muss.

Die Überprüfung, ob diese Bedingungen erfüllt werden, leisten verschiedene Tests, die vom Tool bereit gestellt werden.

Die Aneinanderreihung der einzelnen Properties erfolgt also so, dass das Ende einer Property direkt den Anfang der nachfolgenden Property bildet. Genau genommen könnte damit aus den einzelnen Properties auch eine einzige große Property gebildet werden. Der Vorteil der einzelnen Properties liegt allerdings darin, dass diese auch einzeln überprüft werden können und sich der Nachweis der vollständigen Funktionalität aus der Zusammenset-

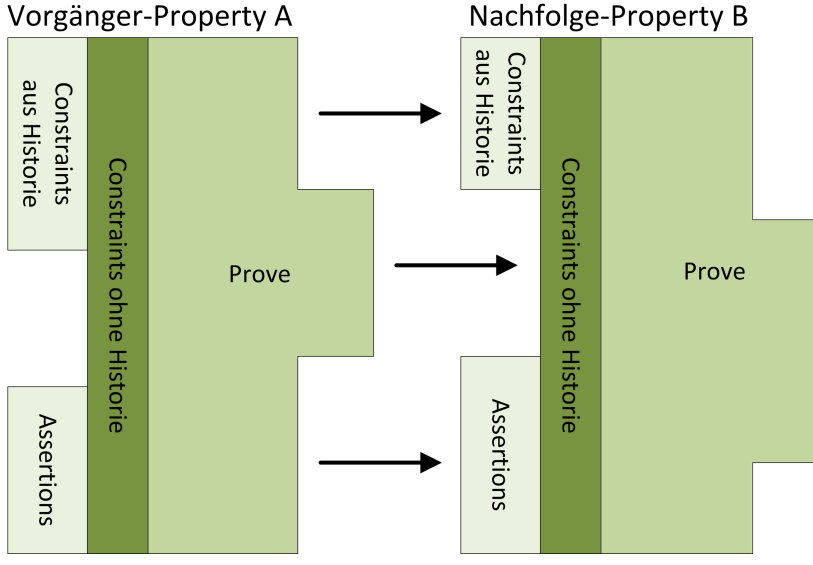


Abbildung 2.11: Verknüpfung von zwei Properties

zung der einzelnen Properties ergibt. Die Verwendung einzelner Properties hat den Vorteil, dass sie im allgemeinen weniger komplex aufgebaut sind, was zum einen die Übersicht für den Verifikateur erhöht, zum andere aber auch die Komplexität für die Verifikation reduziert. Einzelne, einfache Properties können üblicherweise schneller bewiesen werden als eine einzelne komplexere Property.

2.5.3 Completeness Test

Das Tool OneSpin 360® MV bietet verschiedene Tests an, mit denen die Vollständigkeit eines Property-Sets überprüft werden kann. Erst nachdem alle diese Tests erfolgreich bestanden wurden, ist garantiert, dass die Abdeckung der Verifkation vollständig für den definierten Cluster ist. Im folgenden sollen die einzelnen Tests nun kurz beschrieben werden.

Reset Test Für eine erfolgreiche Verifikation ist ein wohl definierter Startzustand nach dem Einschalten oder nach einem Reset erforderlich. Zur Definition dieses Startzustandes muss für jeden Cluster eine Reset-Property existieren, die die Registerbelegung nach einem Reset bestimmt. Sie ist der Ausgangspunkt für alle weiteren Properties in der Kette. Die Reset-Property hat als einzige Annahme die Reset-Sequenz, die wiederum nur aus Input-Signalen bestehen darf. Die Reset-Sequenz selbst wird beim Einlesen der Hardwarebeschreibunsgdateien des Systems definiert. Sie besteht üblicherweise nur aus einem einfachen Wert der Reset-Leitung, kann jedoch auch komplexere Formen annehmen, wenn dies nötig ist.

Neben der Existenz einer Reset-Property überprüft der Rest Test des Completeness Checkers die folgenden Punkte:

1. Die Annahmen dürfen nur Input-Signale enthalten (Reset Assumption Test)
2. Alle Ausgänge müssen determiniert sein (Reset Determination Test)

3. Alle Fälle für einen Reset sind abgedeckt, ein Reset ist immer möglich (Reset Case Split Test)

Successor Test Mit Hilfe des Successor Tests wird überprüft, ob zwei Properties wie im Property Graph angegeben aufeinander folgen können. Dazu wird immer ein Paar aus Vorgänger- und Nachfolger-Property betrachtet. Die Bedingung, dass die Verkettung funktionieren kann, ist die, dass die Annahmen der Nachfolge-Property vollständig durch die Vorgänger-Property bewiesen wurden.

Zu dieser Regel gibt es jedoch zwei Ausnahmen. So sind alle Input-Signale von der Überprüfung durch den Successor Test ausgeschlossen, da sie ja von außen vorgegeben werden und damit nicht vom System direkt beeinflussbar sind. Damit ist es natürlich auch nicht möglich, diese Signale zu determinieren. Ähnlich verhält es sich mit Constraints, die bei den Properties angegeben wurden. Auch diese Annahmen werden als extern vorgegeben angenommen und sind damit nicht beweisbar.

Determination Test Eine weitere Bedingung, die für den Nachweis der vollständigen Verifikation eingehalten wird, ist die lückenlose Determinierung aller Ausgangssignale. Wie schon in Abschnitt 2.5.2 angesprochen, müssen die in der Completeness Beschreibung angegebenen Ausgangssignale zu jedem Zeitpunkt festgelegt sein. Eine Ausnahme davon sind bedingte Determinierungen, bei denen die Signale nur unter bestimmten Randbedingungen festgelegt sein müssen.

Der Determination Test überprüft deswegen, ob die Ausgangssignale innerhalb einer Property immer festgelegt sind. Auch der Determination Test wird wieder an einem Paar von aufeinander folgenden Properties durchgeführt, um zu überprüfen, ob die Anforderungen auch beim Übergang von einer Property zur nachfolgenden eingehalten werden. Damit sollen Doppeldeutigkeiten sowie fehlende Determinierungen am Übergang zwischen den Properties ausgeschlossen werden.

Grundsätzlich muss jedes determinierte Signal immer determiniert sein, dass heißt, zu jedem beliebigen Zeitpunkt muss in jeder beliebigen Property definiert sein, welchen Wert das Signal besitzt. Hängt der Wert eines Signals wiederum von anderen Signalen ab, so müssen auch diese zum jeweiligen Zeitpunkt determiniert sein. Soll der Wert eines Signals unter bestimmten Voraussetzungen nicht festgelegt werden (don't care) so ist dies mit Einschränkungen durch eine so genannte bedingte Determinierung möglich. Hierbei muss das Ausgangssignal nur dann determiniert werden, wenn eine gegebene Bedingung eingehalten wird. Allerdings muss in diesem Fall das bedingende Signal immer determiniert sein.

Case Split Test Ein weiterer Test zur Überprüfung der Vollständigkeit ist der so genannte Case Split Test. Auch hier werden wieder zwei aufeinander folgende Properties miteinander verglichen. Im Gegensatz zum Sucessor Test, beim dem nur überprüft wird, ob eine Property auf eine andere folgenden kann, überprüft der Case Split Test die Vollständigkeit der Nachfolgemöglichkeiten. Mit diesem Test wird also überprüft, ob für alle möglichen Eingangsbelegungen der Vorgängerproperty auch eine Nachfolgerproperty existiert. Im einfachsten Fall bedeutet das, dass eine Property immer einer anderen folgt und ihre Annahmen deswegen vollständig mit den Beweisen in der vorherigen Property übereinstimmen. Es ist jedoch auch möglich, dass als Nachfolger für eine Property mehrere verschiedene Properties in Form einer Fallunterscheidung existieren. Dann muss sicher gestellt werden, dass die Summe dieser Folgeproperties wiederum alle möglichen Kombinationen abdeckt.

Damit stellt dieser Test sicher, dass für jede Kombination von Eingangssignalen auch immer eine Folgeproperty existiert, um die Kette fortzusetzen. Bei Successortest hingegen spielen die Eingangsbelegungen keine Rolle, hier wird nur auf einen passenden Übergang vom Prove zum Assume Teil geachtet.

2.6 Bussysteme im Automobilbereich

Wie auch in anderen Bereichen nimmt die Kommunikation im automobilen Umfeld eine immer wichtigere Rolle ein. Während die ersten Automobile nur sehr wenige elektrische Komponenten wie die Zündung und die Beleuchtung enthalten haben, ist deren Zahl in den letzten Jahren beständig angestiegen. Immer mehr Komfort- und Sicherheitsfunktionen werden durch den Einsatz von immer mehr elektronischen Steuergeräten (Electronic Control Unit (ECU)) realisiert. In den Anfangsjahren stand für jede Funktionalität ein isoliertes Steuergerät mit eigenen Sensoren und Aktoren zur Verfügung. Komplexere Funktionalitäten wie zum Beispiel die automatische Abstandsregelung erfordern jedoch eine steuergeräteübergreifende Zusammenarbeit, um im vorgenannten Beispiel die Entfernungsmessung mit der Ansteuerung des Motors, des Getriebes und der Bremsanlage zu verknüpfen.

Aus der steigenden Komplexität der zu realisierenden Funktionalität ergibt sich also zwangsläufig die Notwendigkeit, die verschiedenen Steuergeräte durch Bussysteme miteinander zu vernetzen. Damit einher geht die Möglichkeit zur Reduktion der Anzahl der verbauten Geräte. Dies ist möglich, weil die ECUs auf in anderen Steuergeräten vorhandene Informationen zurückgreifen können. Ein Beispiel dafür ist die am Rad gemessene Geschwindigkeit, die neben der Anzeige am Tachometer unter anderen auch für das Anti-Blockier-System und die Abstandsregelung verwendet werden kann. Für die Umsetzung der Kommunikation kommen grundsätzlich zwei Varianten in Frage. Die Verwendung von proprietären, herstellerspezifischen Bussen auf der einen Seite und dem Einsatz von standardisierten Bussystemen auf der anderen.

Bei den proprietären Bussystemen handelt es sich um herstellerspezifische Entwicklungen. In der Anfangszeit der Steuergerätevernetzung ging es oft darum, zwei schon vorhandene Geräte aus der eigenen Entwicklung miteinander zu verbinden. Dazu erweiterten die Entwickler die vorhandenen

Geräte durch einfache Kommunikationsprotokolle, die den eigenen Anforderungen genügten. Sie ermöglichen damit einen individuellen Zuschnitt auf die eigenen Anforderungen und können schnell und einfach implementiert werden, da die gesamte Entwicklung unter einem Dach stattfindet.

Solange die Entwicklung von in einem Auto zu verbauenden Steuergeräten alleine vom Hersteller selbst oder ausschließlich mit ihm verbundenen Zulieferern erfolgt, können sich proprietären Protokolle als tauglich erweisen. Die aktuelle Entwicklung geht jedoch dahin, dass einzelne Komponenten von darauf spezialisierten Firmen entwickelt und dann an die OEMs verkauft werden. Diese integrieren die Komponenten dann in ihrem Produkt. Dabei würde es einen nicht unerheblichen Aufwand bedeuten, wenn der Zulieferer seine Komponenten jeweils mit individuellen Schnittstellen für die Kommunikationsprotokolle des aktuell belieferten OEM ausstatten müsste. Der damit verbundene Aufwand würde die Kosten deutlich erhöhen und teilweise sogar unrentabel machen.

Aus diesem Grund gewinnt die Verwendung von standardisierten Bussystemen immer mehr Bedeutung. Solche Bussysteme können von einzelnen Firmen oder auch von übergreifenden Gremien festgelegt werden. Dabei werden in den Standards Festlegungen zu mechanischen Komponenten, der Elektrik, dem Protokollverhalten und weiteren Eigenschaften getroffen. Diese Spezifikationen müssen von den Entwicklern eingehalten werden, um ihr Produkt als konform zu dem jeweiligen Standard zu bezeichnen. Der Vorteil ist, dass damit eine einheitliche, herstellerübergreifende Schnittstelle festgelegt werden kann, die die Zusammenarbeit von Komponenten aus verschiedenen Quellen ermöglicht.

Im Folgenden soll nun kurz der in dieser Arbeit betrachtete Local Interconnect Network (LIN) Bus näher vorgestellt werden.

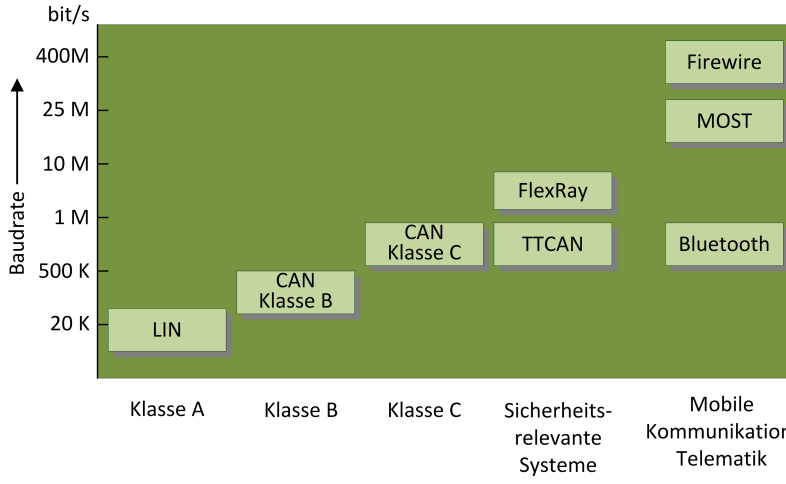


Abbildung 2.12: Klassifikation der Bussysteme im Automobil [10]

2.6.1 Local Interconnect Network (LIN)

Einordnung und Historie Beim Local Interconnect Network (LIN) [14] handelt es sich um ein serielles Busprotokoll. Die erste LIN Spezifikation wurde 1999 veröffentlicht, seit Herbst 2003 ist die überarbeitete Version 2.0 [1] verfügbar. Die aktuellste verfügbare Version ist 2.2A [2] die Ende 2010 veröffentlicht wurde. Hervorgegangen ist das Bussystem aus einem Konsortium der Automobilhersteller Audi, BMW, DaimlerChrysler, Volkswagen und Volvo sowie der Industrieunternehmen Freescale (ehemals Motorola) und Volcano Communications Technologies. Ziel war es, eine kostengünstige Alternative bzw. Ergänzung zum Controller Area Network (CAN) Bus zu schaffen. Dabei wurde besonderes Augenmerk auf eine möglichst einfache und kostengünstige Realisierung der LIN Busteilnehmer gelegt. Geeignet ist der LIN Bus aufgrund seiner geringen Geschwindigkeit insbesondere für einfache

Sensor-Aktor-Systeme und dient so vor allen Dingen der Standardisierung und dem vereinfachten Anschließen von Peripherie.

Eigenschaften Beim LIN Bus handelt es sich um ein „low-cost“ Bussystem zur Verknüpfung von Sensoren und Aktoren mit einem Steuergerät. Es kommt das Single Master - Multiple Slave Konzept zur Anwendung, so dass keine Arbitrierung notwendig ist. Die Implementierung des Busses erfolgt über eine einzelne Datenleitung gegenüber der gemeinsamen Masse des Fahrzeugs als Bezugspotential. Die Signalisierung erfolgt in enger Anlehnung an die UART/SCI Schnittstelle, wobei Open-Collector Treiber mit einem gemeinsamen Pull-Up Widerstand zum Einsatz kommen. Dadurch werden ein dominanter Pegel (logische `'0'`) sowie ein rezessiver Pegel (logische `'1'`) erzeugt. Damit ist eine Realisierung sowohl mit dedizierter Hardware als auch mit vorhandenen Mikrokontrollern in Software möglich. Zur Synchronisierung werden keine zusätzlichen hoch präzisen Quarze in den Slaves benötigt, da die Slaves durch den Master vor jeder Übertragung über den Bus synchronisiert werden. Als maximale Übertragungsgeschwindigkeit sind $20 kbit/s$ definiert. Es sind aber auch niedrigere Geschwindigkeiten erlaubt. Die Kommunikation erfolgt ähnlich wie beim CAN Bus nachrichtenbasiert und enthält damit keine Zieladressierung (Broadcast). Das bedeutet, dass jede Nachricht genau einen Sender aber beliebig viele Empfänger hat.

Im Netzwerk existiert genau ein *Master Node* sowie ein oder mehrere *Slave Nodes*. Alle Teilnehmer beinhalten eine *Slave Task*, die für das Bereitstellen der zu sendenden Daten sowie die korrekte Kommunikation auf dem Bus verantwortlich ist. Zusätzlich besitzt der *Master Node* noch einen *Master Task*, der den zeitlichen Ablauf auf dem Bus steuert (siehe Abbildung 2.13). Eine Übertragung auf dem Bus wird immer von der *Master Task* aus gestartet. Die Slaves dürfen den Bus nicht aktiv belegen, sondern nur auf Anfragen antworten.

Die Kommunikation auf dem LIN Bus erfolgt nach einer im Vorfeld festgelegten Ablauftabelle (*Schedule Table*), die den Buszyklus in mehrere Zeitslots

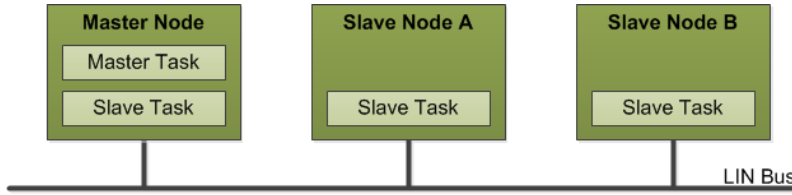


Abbildung 2.13: Struktur des LIN Busses [1]

unterteilt. Für jeden dieser Zeitslots ist festgelegt, welche Nachricht dort gesendet wird. Dadurch wird das Verhalten des Busses deterministisch und echtzeitfähig. Eine Nachricht darf nur in dem ihr zugeordneten Zeitfenster gesendet werden.

LIN Rahmenformat Der für die Kommunikation verwendete Datenrahmen besteht grundsätzlich aus zwei Teilen. Zum einen den *Frame Header*, der immer von der *Master Task* gesendet wird sowie der *Frame Response*, die durch eine *Slave Task* verschickt wird. Dabei kann auch die *Slave Task* des *Master Nodes* diese Antwort schicken und so Daten vom Master zu einem Slave übertragen (siehe Abbildung 2.14).

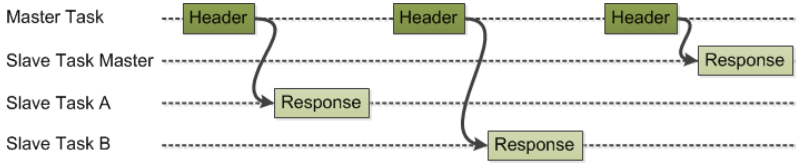


Abbildung 2.14: Ablauf einer LIN Datenübertragung

Die prinzipielle Struktur eines LIN Datenrahmens ist in Abbildung 2.15 zu sehen. Er besteht aus einem *Break-Feld*, gefolgt von vier bis elf Byte-Feldern, die wie in der Abbildung angegeben bezeichnet werden.

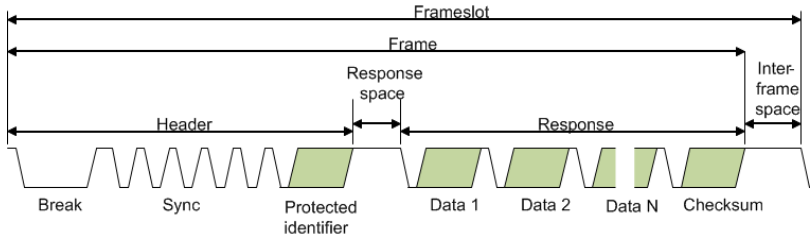


Abbildung 2.15: Struktur eines LIN Datenrahmens

Ein Byte-Feld besteht aus einem dominant kodiertem Startbit, den acht Datenbits und einem rezessiven Stopbit als Abschluss. Bei den Daten wird das niederwertigste Bit zuerst übertragen.

Break-Feld Das *Break-Feld* kennzeichnet den Beginn eines neuen Datenrahmens. Es folgt als einziges nicht der oben abgegebenen Struktur eines Byte-Feldes, da es aus mindestens 13 dominanten Bits, gefolgt von einem rezessiven Bit als Begrenzung, besteht. Das *Break-Feld* darf nur von der *Master Task* gesendet werden. Das *Break-Feld* kann zu jeder Zeit gesendet werden und unterbricht damit unter Umständen auch eine laufende Übertragung. Es muss von den *Slave Tasks* innerhalb von elf Bitzeiten als *Break-Feld* erkannt werden.

Synch-Feld Beim *Synch-Feld* handelt es sich um ein Byte-Feld mit dem Wert `'0x55'`, das so einen Flankenwechsel pro Bit auf dem Bus erzeugt, der von den Slaves für die Synchronisierung auf die Datenraten des Masters verwendet wird.

Protected Identifier (PID) Ein Protected Identifier (PID) besteht zum einen aus dem Identifier selbst (Bits 0 bis 5) sowie zwei Paritätsbits zur Fehlererkennung (Bits 6 und 7). Der Identifier dient der Unterscheidung des Typs

und des Inhaltes einer Nachricht. Durch ihn wird auch festgelegt, welche *Slave Task* auf den *Frame Header* antworten muss. Für den Identifier werden die folgenden Bereiche unterschieden:

Identifier	*Bedeutung*
0-59	Message Identifier
60, 61	Konfiguration und Diagnose
62	OEM-spezifische Kommunikation
63	Reserviert für zukünftige Erweiterungen

Die Paritätsbits werden aus den Bits des Identifiers anhand der Formeln 2.1 und 2.2 berechnet.

$$P0 = ID0 \oplus ID1 \oplus ID2 \oplus ID4 \tag{2.1}$$

$$P1 = \overline{ID1 \oplus ID3 \oplus ID4 \oplus ID5} \tag{2.2}$$

Daten Ein Frame kann zwischen einem und acht Datenbytes enthalten. Die genaue Anzahl hängt von dem gesendeten *Protected Identifier* ab und muss im Vorfeld bei der Konfiguration des Busses für alle Teilnehmer konfiguriert werden. Ein Datenbyte wird dabei mit einem Byte-Feld übertragen. Bei mehreren Bytes wird zuerst das Byte mit dem niederwertigsten Bit übertragen.

Checksumme Die Checksumme berechnet sich als invertierte acht Bit breite Summe mit Übertrag aus den gesendeten Datenbytes, beziehungsweise der Datenbytes und des *Protected Identifiers*. Dies entspricht der Addition der Bytes und der Subtraktion von 255, sobald die Summe den Wert 256 oder größer erreicht hat.

Werden nur die Datenbytes in die Checksumme mit einbezogen, so wird dies als *Classic Checksum* bezeichnet. Sie wird für die Kommunikation nach dem LIN Standard 1.3 verwendet. Erstreckt sich die Berechnung der Checksum-

me zusätzlich auch auf den PID, so bezeichnet man sie als *Enhanced Checksum*, die bei LIN 2.0 angewandt wird.

Eine Ausnahme davon stellen die Identifier 60 bis 63 dar, bei denen immer die *Classic Checksum* zum Einsatz kommt.

Frame-Typen Die LIN Spezifikation sieht verschiedene Typen von Frames vor, die unter verschiedenen Randbedingungen zum Einsatz kommen können:

Unconditional Frame Der *Unconditional Frame* wird für die normale Datenübertragung verwendet. Wenn in der *Schedule Table* ein entsprechender Slot vorgesehen ist, sendet der *Master Task* einen entsprechenden Header. Daraufhin muss genau eine angesprochene *Slave Task* mit der Response antworten. Diese Antwort kann auch von beliebigen anderen *Slave Tasks* empfangen und an die Anwendung weiter gereicht werden, wenn keine Fehler bei der Übertragung entdeckt wurden.

Event Triggered Frame Wenn nur selten Daten von einzelnen *Slave Tasks* abgerufen werden sollen ist es ineffizient, dafür einen festen Slot in der *Schedule Table* zu reservieren, da dadurch die für andere Übertragungen zur Verfügung stehende Bandbreite reduziert wird.

Für diesen Anwendungsfall können *Event Triggered Frames* verwendet werden. Dazu wird ein Header mit mehreren Responses verknüpft, die von unterschiedlichen *Slave Tasks* bereit gestellt werden. Eine *Slave Task* darf dabei nur auf eine Anfrage antworten, wenn sich seine zu übermittelnden Daten geändert haben. Das bedeutet, dass unter Umständen auch gar keine Antwort gesendet wird.

Wenn mehre *Slave Tasks* gleichzeitig eine Antwort senden, kann es zu einer Kollision kommen. Diese muss vom Master dadurch aufgelöst werden, dass

er die Daten nach der Kollisionserkennung sukzessive mit *Unconditional Frames* abfragt.

Sporadic Frame Mit *Sporadic Frames* soll die Möglichkeit geschaffen werden, dynamisches Verhalten in den strengen Rahmen der deterministischen *Schedule Table* zu bringen. In dem für *Sporadic Frames* vorgesehenen Zeitfenster wird immer dann von der *Master Task* der entsprechende Header gesendet, wenn sie weiß, dass sich eines der damit verbundenen Signale geändert hat.

Wenn mehrere Nachrichten mit einem Zeitslot verbunden sind, wird die Nachricht mit der höchsten Priorität zuerst gesendet. Wenn es bei keiner der verknüpften Nachrichten Änderungen gegeben hat, findet in dem Zeitfenster gar keine Übertragung statt.

Da die *Master Task* wissen muss, ob sich ein Signal in einem *Sporadic Frame* geändert hat, ist sie normalerweise auch diejenige, die die Nachrichten versendet.

Diagnostic Frames Mit *Diagnostic Frames* werden immer Diagnose- oder Konfigurationsdaten übertragen. Die Nachricht enthält dabei immer genau acht Datenbytes. Die genaue Interpretation der übertragenen Daten wird in der *LIN Diagnostic and Configuration Specification* [1] festgelegt.

User-defined Frames Der Inhalt von *User-defined Frames* ist beliebig und wird vom Anwender festgelegt. Der entsprechende Header muss immer dann gesendet werden, wenn der zugeordnete Zeitslot bearbeitet wird.

Netzwerk Management Das *Netzwerk-Management* beschränkt sich beim LIN Bus allein auf das Aufwecken des Clusters aus dem Ruhezustand (*Wake Up*) und das Einnehmen des Ruhezustandes (*Goto Sleep*).

Jeder Node in einem schlafenden LIN Cluster kann einen *Wake Up* anfordern. Dazu wird der Bus für die Dauer von 250μs bis 5ms auf den dominanten Pegel gezogen. Alle anderen Knoten müssen dieses Signal erkennen und den Ruhezustand innerhalb von 100ms verlassen haben.Der Master wacht ebenso auf und beginnt das Versenden von *Frame Headern*, um die Ursache des Aufweckens heraus zu finden.

Alle Nodes in einem aktiven Cluster können durch das Versenden eines speziellen *Diagnostic Frames* in den Ruhezustand versetzt werden. Dies geschieht auch automatisch, wenn der Bus länger als 4s inaktiv ist.

3 Problemstellung und Stand der Technik

3.1 Zielsetzung

Die funktionale Verifikation ist ein wichtiger Schritt bei der Entwicklung komplexer integrierter Schaltungen. Durch sie wird sichergestellt, dass die entwickelte Schaltung auch den ursprünglichen Spezifikationen entspricht. Während die Anzahl der auf einem Chip integrierbaren Funktionen immer noch nach Moores Law [16] wächst, kann die Produktivität beim Entwurf und der Verifikation der Schaltungen damit nicht mehr mithalten. Es entsteht damit das Design- und Verification Gap (vergl. 3.1). Die komplexen Schaltungen, die technisch gefertigt werden könnten, lassen sich nicht mehr ausreichend verifizieren, um einen fehlerfreien Betrieb sicher zu stellen. Hier sind neue Ansätze und Methoden erforderlich, um eine Verbesserung der Qualität und der Aussagekraft der Verifikation integrierter Schaltungen zu erreichen. Die vorliegende Arbeit liefert einen Beitrag zur Verbesserung der Verifikationsqualität.

Eine Möglichkeit zur Verbesserung der Verifikationsqualität stellt der Einsatz von formalen Verifikationsmethoden dar. Simulative Ansätze lassen im Allgmeinen keine definitiven Aussagen zur Fehlerfreiheit eines Systems zu. Statt dessen kann mit ihrer Hilfe immer nur das Vorhandensein eines Fehlers nachgewiesen werden. Im Gegensatz dazu ermöglicht die formale Verifikation sehr wohl die Aussage, dass ein überprüftes Design fehlerfrei im Sinne der Spezifikation ist. Ein Beitrag dieser Arbeit liegt deswegen in der Vorstellung einer Methodik zum Einsatz formaler Techniken für die Verifikation elektronischer Schaltungen.

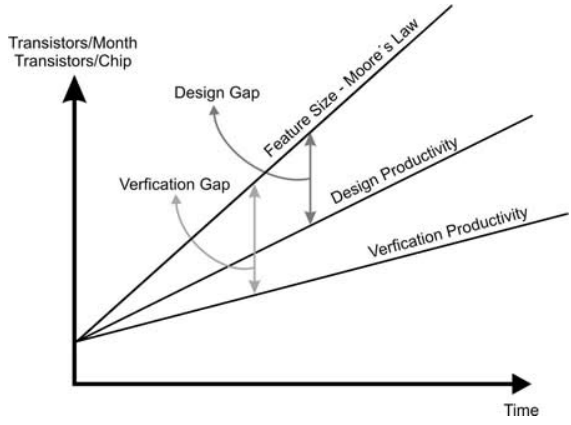


Abbildung 3.1: Design and Verification Gap (aus: [15])

Eine der fundamentalen Forderungen zur Sicherstellung der Qualität bei der Verifikation von integrierten Schaltungen ist die strikte Trennung zwischen dem Entwurf der Schaltung und der Verifikation. Im Idealfall werden diese beiden Tätigkeiten von unterschiedlichen Personen durchgeführt, die sich bei ihrer Arbeit jeweils nur auf die zu Grunde liege Spezifikation beziehen, ohne Kenntnis von den Details der Arbeit des Anderen zu haben. Damit soll vermieden werden, dass zum Beispiel für die Verifikation die Spezifikation schon im Sinne der Realisierung interpretiert wird. Mit diesem Wissen bestünde die Gefahr, die Testfälle schon auf die zu testende Hardware abzustimmen und damit schon von der Spezifikation abzuweichen. Auch umgekehrt besteht die Gefahr, dass ein Hardwareentwickler sein System genau so entwirft, damit ein ihm bekannter Testfall erfüllt wird. Um diese Fehlerquellen zu minimieren, soll hier ein Ansatz vorgestellt werden, der eine weitgehende Entkopplung des Hardwareentwurfs von dessen Verifikation ermöglicht.

Wie schon weiter oben beschrieben, fällt ein nicht unerheblicher Teil des Entwicklungsaufwandes für eine elektronische Schaltung für deren Verifikation

an. Aus diesem Grund erscheint es wünschenswert, wenn möglichst viel dieses Aufwandes für ein anderes Projekt wiederverwendet werden kann. Dies ist besonders interessant für Komponenten zur Kommunikation über ein standardisiertes Kommunikationsprotokoll. Beispiele dafür sind Buscontroller und Bridges. Hier ist es erstrebenswert, die Funktionalität des Kommunikationsprotokolls unabhängig von der tatsächlichen Implementierung zu beschreiben. Diese einheitliche Beschreibung des Sollverhaltens nach Maßgabe des Kommunikationsprotokolls sollte dann auf unterschiedliche Implementierungen angewendet werden können, um deren Konformität nachzuweisen.

Bei der Betrachtung von Kommunikationsprotokollen sind zwei grundsätzliche Varianten zu unterscheiden, die parallele und die serielle Kommunikation. Bei der parallelen Datenübertragung werden die zu übertragenden Daten und eventuell benötigte Steuerinformationen über individuelle Leitungen parallel und quasi gleichzeitig übertragen. Ein Übertragungszyklus dauert damit nur einige wenige Taktzyklen, was das zu betrachtende Zeitfenster bei der formalen Verifikation entsprechend klein hält.

Deutlich anders ist die Situation bei der seriellen Übertragung. Hier steht nur eine einzige Signalleitung zur Verfügung, über die sowohl Daten als auch Steuerinformationen übertragen werden müssen. Da pro Zeitschritt bei zweiwertigen Signalen nur jeweils ein Bit übertragen werden kann, dauert ein Übertragungszyklus hier deutlich länger und kann mehrere Dutzend bis einige Tausend Taktschritte benötigen. Im Gegensatz zur parallelen Übertragung, bei der alle benötigten Informationen direkt aus den verschiedenen Signalleitungen verfügbar sind, muss bei der seriellen Übertragung die Bedeutung aus der Historie der Übertragung entnommen werden. Für die Verifikation solcher seriellen Protokolle sind also im Allgemeinen wesentlich größere Zeitfenster zu betrachten, was den Aufwand deutlich erhöht. Hier liefert die vorliegende Arbeit eine Methode zur effizienten Anwendung von formaler Verifikation auf serielle Protokolle.

Der Nachweis der korrekten Implementierung des Kommunikationsprotokolls stellt sicher, dass es zu keinen Störungen der Kommunikation durch einzelnen Teilnehmer kommen kann. Nicht nachgewiesen ist damit allerdings die korrekte Übertragung von Daten an sich, also der Ende-zu-Ende Transfer vom Businterface zur Hostschnittstelle. So kann ein Teilnehmer zwar durchaus das Kommunikationsprotokoll einhalten, bei der internen Verarbeitung der Daten aber Fehler machen. Aus diesem Grund wird in der vorliegenden Arbeit auch die korrekte Datenverarbeitung betrachtet.

Die Hauptziele dieser Arbeit sind also:

1. Die Steigerung der Aussagekraft zur Protokollkonformität durch Anwendung Formaler Verifikation
2. Die Entkopplung zwischen Verifikateur und Entwickler
3. Eine vereinfachte Wiederverwendung des Propertysets für andere Cores
4. Die Verifikation langer serieller Protokolle
5. Die Betrachtung der Ende zu Ende Datenübertragung

3.2 Anwendungsbereich

Aktuelle Kommunikationssysteme im Automobilbereich bestehen aus einer Vielzahl einzelner Komponenten, die erst in Ihrer Gesamtheit die gewünschte Funktionalität bereit stellen. In der Abbildung 3.2 ist ein solcher stark vereinfachter Systemaufbau zu sehen.

Wie in der Abbildung zu erkennen ist, ergibt sich ein hierarchischer Aufbau des Systems. Auf der unteren Ebene befinden sich einzelne Intellectual Property Cores (IP-Cores), die Basisfunktionalitäten bereit stellen. Neben Prozessoren und Speicherblöcken sind das unter anderem auch IP-Blöcke, die für die Verarbeitung von Kommunikationsprotokollen zuständig sind. Diese einzelnen Blöcke werden im Allgemeinen von verschiedenen Zuliefe-

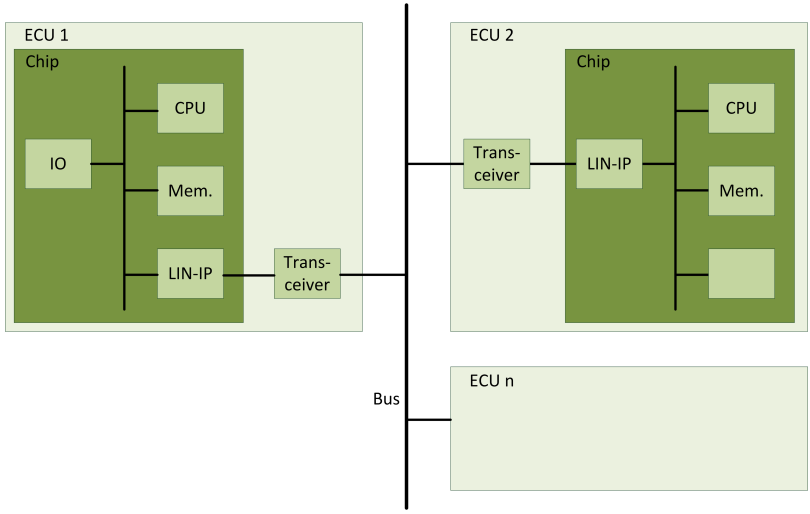


Abbildung 3.2: Schematische Darstellung eines Kommunikationssystems

rern bereit gestellt und von einem Hersteller zu einem Chip integriert. Dieser Chip wiederum ist eine Komponente von mehreren, die von einem weiteren Hersteller auf einer gedruckten Schaltung zu einer ECU integriert werden. Der OEM als eigentlicher Produzent eines Automobils integriert wiederum ECUs unterschiedlicher Zulieferer in sein Produkt. Das so zusammengesetzte System ist also hochgradig heterogen, was die Herkunft der einzelnen Teilkomponenten angeht. Insbesondere hat der OEM unter Umständen nur wenig bis gar keine direkten Kenntnisse von den auf den unteren Ebenen eingesetzten Komponenten.

3.2.1 Zulieferpyramide

Aus dem im vorherigen Abschnitt beschriebenen Systemaufbau ergibt sich im Automobilbereich eine pyramidenförmige Struktur aus einzelnen Lieferanten bis hin zum OEM, die aus diesem Grund auch als Zulieferpyramide bezeichnet wird (Abbildung 3.3).

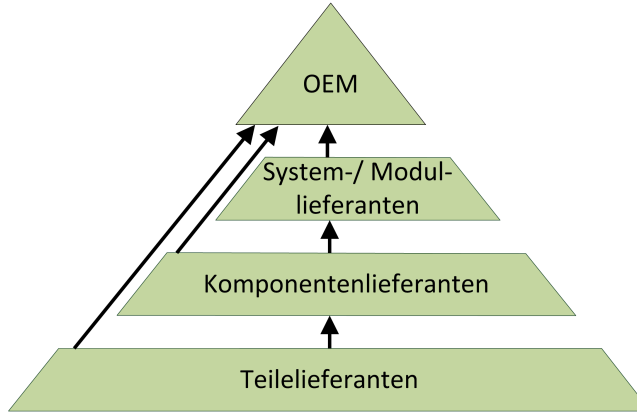


Abbildung 3.3: Lieferantenpyramide (nach [21])

In dieser Pyramide werden verschiedene Akteure unterschieden:

OEM - Hersteller. Der OEM (Original Equipement Manufacturer) ist der Hersteller des Automobils. Er vergibt Unteraufträge zur Entwicklung und Produktion an weitere Zulieferer. Dazu übergibt er Koordinierungsaufwände an den Tier-1, der wiederum weitere Lieferanten beauftragen kann. Der OEM stellt das vollständige System aus den Einzelkomponenten zusammen.

Tier-1 Zulieferer. Der Tier-1 ist Lieferant eines Teilsystems. Er wurde dazu direkt vom OEM beauftragt und entwickelt und fertigt die Komponenten nach den Vorgaben des OEM.

Tier-n Zulieferer. Neben dem Tier-1 können weitere Unterauftragsnehmer existieren, die nicht direkt vom OEM, sondern von einen der anderen Unterauftragnehmer beauftragt wurden. Geordnet nach ihrer Entfernung zum OEM werden diese als Tier-2 bis Tier-n bezeichnet.

Durch diese Aufteilung von Teilleistungen eines Gesamtsystems auf mehrere Zulieferer ist es besonders wichtig, die Schnittstellen zwischen den einzelnen Komponenten sowie deren geforderte Funktionalität klar zu definieren und vor allem auch zu überprüfen. Dies ist eine Grundvoraussetzung, um die fehlerfreie Zusammenarbeit aller Komponenten zu ermöglichen. Dabei spielen die IP-Cores, die für die externe Kommunikation eingesetzt werden eine bedeutende Rolle. Sie sind essentiell für die Kommunikation und die fehlerfreie Zusammenarbeit zwischen den einzelnen ECUs eines Systems im Automobil. Alle Daten, die zwischen den einzelnen Steuergeräten ausgetauscht werden, laufen über diese IP-Cores. Ein Fehler an dieser Stelle wirkt sich damit mit großer Wahrscheinlichkeit negativ auf das Gesamtsystem aus und kann es unter Umständen komplett lahmlegen. Bedingt durch ihre Lage tief in der Hierarchie finden deren Tests relativ früh im Entwicklungsprozess statt. Die durch sie verursachten Fehler hingegen werden unter Umständen erst gegen Ende der Entwicklung in der Integrationsphase sichtbar. Die Behebung eines Fehlers, der erst zu diesem Zeitpunkt entdeckt wird, ist sehr aufwändig und kann zu sehr hohen Kosten und Verzögerungen führen.

Aus diesen Gründen ist es wünschenswert, die korrekte Funktionalität der Kommunikationskomponenten schon möglichst früh im Entwurfsprozess

sicherstellen zu können. Dazu werden üblicherweise komplexe Testbenches eingesetzt, die die Komponente mit verschiedenen Testfällen beaufschlagen und die Reaktion der Komponente überprüfen. Der Nachteil bei einer solchen simulationsbasierten Untersuchung ist jedoch, dass mit Hilfe einer Simulation im Allgemeinen nur das Vorhandensein eines Fehlers, nie jedoch aber dessen Abwesenheit nachgewiesen werden kann. Für einen vollständigen Nachweis müssten mit der Simulation sämtliche möglichen Eingangskombinationen bei allen möglichen internen Zuständen betrachtet werden. Dies ist jedoch bei den heute verwendeten komplexen Systemen praktisch unmöglich, da die benötigte Simulationszeit viel zu groß wird. Somit bleibt man auf eine reduzierte Aussagekraft begrenzt. Sie kann durch geschickte Auswahl der Testfälle erhöht werden, wird in der Praxis aber nie einen Beweis liefern.

Hier bietet die Verwendung der formalen Verifikation einen Ansatz, um dieses Problem zu lösen. Im Gegensatz zur Simulation sind mit der formalen Verifikation Beweise zum Verhalten eines Systems möglich. Mit ihr lässt sich die Gültigkeit einer Eigenschaft für alle möglichen Fälle nachweisen. Wenn es also gelingt, das in der Spezifikation geforderte Verhalten in Eigenschaftsbeschreibungen abzubilden und diese erfolgreich an einer zu überprüfenden Komponenten anzuwenden, so kann damit deren Spezifikationskonformität nachgewiesen werden. Unter der Annahme, dass die Spezifikation selbst fehlerfrei ist, sollte es somit ausgeschlossenen sein, dass diese Komponente im Zusammenspiel mit weiteren Einheiten zu einem Fehlerverhalten im Gesamtsystem führt. Mögliche Fehler werden damit schon früh im Entwicklungsprozess ausgeschlossen.

3.3 Besonderheiten bei der Verifikation serieller Protokolle

Bei der Verifikation serieller Protokolle spielen deutlich andere Randbedingungen eine Rolle, als das bei paralleler Kommunikation oder der Verifikation von Rechenwerken der Fall ist. Bei paralleler Kommunikation werden

im Allgemeinen Steuerinformationen und Daten auf dedizierten Leitungen übertragen. Im Idealfall ist eine Information einer Leitung fest zugewiesen. Damit sind diese Informationen jederzeit direkt ablesbar. Eine Übertragung selbst ist je nach Breite des Datenbusses innerhalb weniger Taktzyklen abgeschlossen. Aus der größeren Anzahl von Taktschritten, die betrachtet werden müssen, ergeben sich bei der Verifikation serieller Protokolle erheblich größere Zeitfenster. Mit der Größe des zu betrachtenden Zeitfensters steigt auch die Komplexität und die Laufzeit des Beweises.

Im Gegensatz dazu müssen bei seriellen Protokollen die Informationen über eine einzige Leitung übertragen werden. Da pro Zeitschritt nur ein einziges Bit (bei zweiwertigen Codierungen) übertragen werden kann, beläuft sich die Länge einer Übertragung schnell auf einige Dutzend bis zu mehreren Tausend Bits. Erschwerend kommt hinzu, dass es im Normalfall keine getrennten Leitungen für Steuerinformationen und Daten gibt. Beide werden abwechselnd über die gleich Leitung übertragen. Eine Unterscheidung zwischen Daten und Steuerinformationen ist damit nur über die Verfolgung des zeitlichen Ablaufes möglich. Erst damit kann den einzelnen Bits ihre korrekte Bedeutung zugeordnet werden. Dadurch ist die Kenntnis der Historie einer Übertragung von entscheidender Bedeutung für die korrekte Interpretation.

3.4 Trennung von Verifikation und Implementierung

Ziel der Verifikation ist es, die korrekte Implementierung der Spezifikation nachzuweisen. Dazu ist es wünschenswert, wenn diese Überprüfung von anderen Personen vorgenommen wird als von denjenigen, die die Implementierung erstellt haben. Auch in der heutigen Zeit werden die meisten Spezifikationen per Hand von Menschen erstellt und in natürlicher Sprache beschrieben. Zusammen mit der immer weiter steigenden Komplexität der Protokolle ergeben sich damit einige Probleme, die sich bei der Verifikation bemerkbar machen. Am augenscheinlichsten ist die Tatsache, dass sich ein

Text in natürlicher Sprache oft auf verschiedene Arten interpretieren lässt. Die Intention desjenigen kann eine völlig andere gewesen sein, als sie von demjenigen interpretiert wird, der eine Umsetzung der Spezifikation vornimmt. Wenn nun die gleiche Person, die die Implementierung vornimmt auch für die Verifikation verantwortlich ist, besteht die große Gefahr, dass solche Fehlinterpretationen nicht entdeckt werden. Der Entwickler wird seine Interpretation der Spezifikation selbstverständlich auch für die Verifikation der Implementierung anwenden und damit solche Fehler gar nicht entdecken. Aus diesem Grund erscheint es mehr als angeraten, dass die Implementierung und die Verifikation von unabhängigen Personen durchgeführt werden. Sollten Teile der Spezifikation unterschiedlich interpretiert werden, so resultiert das in einem Fehler bei der Verifikation. Entwickler und Verifikateur haben dann die Möglichkeit, gemeinsam ihre Interpretation zu vergleichen und zu korrigieren. Eine Garantie für ein korrektes Verhalten im Sinne der Spezifikation ist dann allerdings immer noch nicht gegeben. Dazu wäre es nötig, die fraglichen Stellen auch mit der Person zu besprechen, die die Spezifikation verfasst hat. Gerade bei öffentlichen, von einem großen Konsortium verfassten Standards ist das aber keine praktikable Möglichkeit.

Eine weitere Fehlerquelle ist die Tatsache, dass eine natürlichsprachliche Spezifikation, nicht immer vollständig ist. Diese Freiheiten können dann vom Entwickler genutzt werden, indem er dafür das Verhalten seiner Implementierung definiert. Da dabei nicht gewährleistet ist, dass sich diese Interpretation nicht negativ auf das Verhalten der Implementierung auswirkt, sollten auch solche Stellen nach Möglichkeit erkannt werden. Ebenso bietet wie bei der Fehlinterpretation von Informationen auch wieder die Überprüfung durch eine andere unabhängige Person die Möglichkeit, solche offene Stellen in der Spezifikation zu entdecken.

Eine weitere Fehlerquelle bei der Verifikation stellt auch eine zu detaillierte Kenntnis der aktuellen Implementierung dar. So wird die Spezifikation auch von einer anderen Person unter Umständen so interpretiert und getestet, wie es die dem Verifikateur bekannte Implementierung nahelegt. Damit bleiben

andere Fehlerquellen jedoch unter Umständen unberücksichtigt. Auf der anderen Seite legt die Kenntnis des Testsystems für den Entwickler auch eine bestimmte Art der Implementierung nahe, indem er versucht, seine Implementierung so zu gestalten, dass sie die Tests erfolgreich bestehen kann.

Zusammengefasst sollten die Erstellung der Implementierung und die Durchführung der Verifikation immer von getrennten Personen durchgeführt werden, die jeweils möglichst wenig Informationen über die Details der Arbeit des anderen haben sollten. Nur so kann erreicht werden, dass die Spezifikation zweimal unabhängig von einander interpretiert wird. Dies ist erforderlich, um Fehler und Ungenauigkeiten in der Spezifikation entdecken zu können. Auch wird damit vermieden, dass die Implementierung so ausgeführt wird, dass sie die bekannten Tests bestehen kann und umgekehrt.

Neben der Erhöhung der Testqualität durch den Einsatz von zwei unterschiedlichen Personen für die Implementierung und die Verifikation gibt es auch noch einen weiteren Grund für eine solche Aufteilung. Damit ist es zusätzlich auch möglich, dass die Verifikation zeitgleich mit der Implementierung angegangen wird. Im Idealfall kann die Implementierung gleich nach deren Fertigstellung überprüft werden. Damit ergibt sich ein nicht zu unterschätzender Zeitvorteil gegenüber einer rein sequentiellen Vorgehensweise. Im Idealfall werden Implementierung und Verifikation schon früh aufeinander angewendet und iterativ immer weiter verfeinert.

Ein weiterer Vorteil der Unabhängigkeit von Implementierung und Verifikation ist auch die dadurch potentiell höhere Wiederverwendbarkeit der Verifikationsumgebung. Je weniger Detailwissen zu der zu testenden Implementierung in die Verifikationsumgebung eingeflossen ist, desto größer ist die Wahrscheinlichkeit, dass diese auch direkt für anderen Implementierungen der gleichen Spezifikation verwendet werden kann. Im Idealfall sind nur kleine Anpassung zum Beispiel hinsichtlich der in der Implementierung verwendeten Benennungen nötig.

3.5 Stand der Forschung

3.5.1 Nguyen: System-on-Chip Protocol Compliance Verification Using Interval Property Checking

Die Arbeit von Duc-Minh Nguyen [17] befasst sich mit der Anwendung von IPC zur formalen Eigenschaftsprüfung von Kommunikationsmodellen. Durch das beim IPC verwendete Berechnungsmodell, kann es zum Auftreten von falschen Gegenbeispielen, sogenannten *false negatives* oder *false counterexamples* kommen. Diese rühren daher, dass Zustände angenommen werden, die vom Anfangszustand aus nicht erreichbar sind. Das bedeutet, dass diese Zustände in einem korrekt initialisierten System nie auftreten können. Diese nicht erreichbaren Zustände werden deswegen nicht erkannt, da immer nur beschränkte Zeitintervalle, ausgehend von einem beliebigen Startzustand, betrachtet werden. Globale Informationen, die zum Ausschluss von nicht erreichbaren Zuständen führen würden, sind hier unter Umständen nicht verfügbar. Um solche falschen Gegenbeispiele zu vermeiden, müssen zusätzliche globale Erreichbarkeitsinformationen in Form von sogenannten Invarianten zu den zu beweisenden Eigenschaften hinzugefügt werden. Die manuelle Ermittlung solcher Invarianten ist jedoch ein aufwändiger Prozess. Hier stellt die Arbeit von Nguyen eine Methode vor, wie solche Invarianten für Kommunikationsmodule automatisch erstellt werden können.

Dazu muss per Hand eine so genannte *main-FSM* erstellt werden, die das Gesamtverhalten der Implementierung steuert. Diese *main-FSM* ist dann die Basis für den von Nguyen entwickelten Traversierungsalgorithmus, der Erreichbarkeitsinformationen für die Zustände der *main-FSM* errechnen kann.

Einen weiteren Teil der Arbeit von Nguyen stellt eine Methodik zur Erstellung vom implementierungsunabhängigen Eigenschaftssätzen dar, die für die Verifikation von Kommunikationsmodulen verwendet werden können. Damit soll ein Eigenschaftssatz für die Verifizierung von verschiedenen Implementierungen verwendet werden können. Grundlage dieser Methodik ist

eine sogenannte Recorder-FST (FST = *finite state transition structure*). Diese beobachtet des Betrieb des Busses und stellt zu jedem Zeitpunkt den Zustand des Busses fest. Sie dient jedoch nicht dazu, fehlerhaftes Verhalten der Implementierung aufzudecken. Zu diesem Zweck kommt der Eigenschaftssatz zum Einsatz, der das Soll-Verhalten der Implementierung unter Einbeziehung des Buszustandes bestimmt.

Für die Verifikation wird die Recorder-FST mit der Implementierung zu einem gemeinsamen Verifikationsmodell (engl. Design Under Verification (DUV)) verbunden. Für dieses DUV kann dann mittels des zuvor erstellten Eigenschaftensatzes das korrekte Verhalten überprüft werden. Für die Erzeugung des Eigenschaftensatzes kommt die weiter oben beschrieben Methodik zur Bestimmung von Invarianten zum Einsatz.

Auch die in dieser Arbeit vorgestellte Methodik verwendet eine FSM zur Beschreibung des korrekten Protokollverhaltens. Im Gegensatz zur *main-FSM* in der Arbeit von Nguyen wird sie allerdings nicht aus der Implementierung, sondern aus der Spezifikation abgeleitet. Die ist wie in Kapitel 3.4 beschrieben, wünschenswert um zu erreichen, dass auch wirklich gegen die Spezifikation und nicht gegen die aktuelle Implementierung verifiziert wird. Somit wird eine saubere Trennung zwischen den Aufgaben des Entwicklers und des Verifikateurs gefördert.

Ähnlich wie die Arbeit von Nguyen zielt auch die vorliegende Arbeit auf einen Eigenschaftssatz zur Verifikation von Kommunikationsprotokollen ab, der unabhängig von der aktuellen Implementierung ist. Mit dem Ansatz von Nguyen ist jedoch nur die Überprüfung des korrekten Protokollverhaltens auf dem Bus möglich. Dazu werden nur Kontrollsignale in die Überprüfung einbezogen. Die Daten werden nicht überprüft. Die vorliegende Arbeit hingegen stellt auch die korrekte Ende-zu-Ende Übertragung der Daten vom Bus zu den internen Ausgängen des Kommunikationsmoduls sicher.

Die Recorder-FST in der Arbeit von Nguyen ist zudem darauf ausgelegt, dass die Steuersignale des Busses direkt abgegriffen werden können, um daraus den aktuellen Zustand des Busses zu bestimmen. Die ist bei paralle-

len Bussen mit separaten Steuerleitungen möglich, nicht jedoch bei seriellen Systemen, bei denen sich Steuerinformationen und Daten die gleiche Leitung teilen. Die in dieser Arbeit vorgestellte Methodik wurde jedoch speziell darauf hin angepasst, um bei seriellen Bussystemen angewendet zu werden.

3.5.2 Yang et.al., Formal Compliance Verification of Interface Protocols

In [22] beschreiben Yang et. al. die Überprüfung eines Kommunikationsmoduls auf korrekte Funktionalität mittels einer sogenannten *spec FSM*. Dazu muss die Spezifikation des Kommunikationsprotokolls per Hand in eine FSM überführt werden, in der für jeden Zustand des Busses festgelegt wird, unter welchen Bedingungen in welchen anderen Zustand gewechselt wird. Neben den erlaubten Zuständen des Busses werden zusätzlich noch zwei weitere Zustände eingeführt. Der Erste dieser beiden Zustände (Violation) wird eingenommen, wenn eine verbotene Signalkombination auf dem Bus erscheint. Er kennzeichnet damit einen Fehler. Tritt hingegen eine Signalkombination auf, die so in der Spezifikation nicht vorgesehen ist, wird der zweite zusätzliche Zustand (dont care) eingenommen. Mit der Einführung dieser *spec FSM* lässt sich dann das korrekte Verhalten eines Kommunikationsmoduls definieren. Es ist dann korrekt, wenn keine Folge von Eingangssignalen existiert, bei der der Zustand Violation eingenommen wird. Um nun die Einhaltung der Spezifikation bei einer Implementierung zu überprüfen, vergleichen Yang et. al. die FSM der Implementierung mit der *spec FSM*. Da die beiden FSM nicht zwangsläufig den gleichen Aufbau haben, kann keine direkte Zuordnung von Zuständen der *spec FSM* zu der FSM der Implementierung vorgenommen werden. Um dennoch eine Äquivalenz nachweisen zu können, erstellen sie einen Zuordnungsbaum aus Zuständen der *spec FSM* und der implementierten FSM. Wenn in diesem Baum keine Kombination aus einem Zustand der Implementierung mit dem Fehlerzustand der *spec FSM* existiert, wird die Implementierung als fehlerfrei angesehen.

Die Arbeit von Yang et. al. verwendet auch eine FSM zur Darstellung des Protokolls aus der Spezifikation, die per Hand zu erstellen ist. Im Gegensatz zu der vorliegenden Arbeit wird diese FSM dann allerdings nicht mit dem HDL Modell der Implementierung, sondern mit einer weiteren FSM verglichen. Um valide Aussagen über die Implementierung treffen zu können, sollte diese FSM direkt aus der Realisierung gebildet werden. Zu diesem Vorgang werden in der Arbeit von Yang et. al. allerdings keine Aussagen getroffen. Unter Umständen kommt also nur eine FSM zur Anwendung, die als Grundlage für die Implementierung diente und per Hand in HDL Code umgesetzt wurde. Damit ist durch den Vergleich aber nicht nachgewiesen, dass der HDL Code der Implementierung tatsächlich korrekt ist.

3.5.3 Kimmeskamp et. al., Formale Verifikation eines komplexen seriellen Kommunikationsprotokolls - „Lessons Learned“ am Beispiel einer FlexRay-IP-Verifikation

Ein weiteres Beispiel für die Verifikation eines seriellen Protokolls wird in [12] beschrieben. Kimmeskamp et. al. betrachten dabei das FlexRay-Protokoll als Beispiel für ein komplexes serielles Kommunikationsprotokoll. Aus Gründen der Komplexität des FlexRay-Protokolls wurde dabei bewusst von einem Blackbox-Ansatz abgesehen und statt dessen einzelne Komponenten der Implementierung separat überprüft. Die getrennt verifizierten Module sollten dann zur Gesamtfunktionalität zusammengesetzt werden.

Ein Nachteil dieses modulbasierten Ansatzes ist, dass fast zwangsweise sehr nahe an der konkreten Implementierung des Protokolls gearbeitet werden muss, da eben genau einzelne Blöcke des Designs verifiziert werden. Damit läuft man als Verifikateur aber auch Gefahr, sich beim Erstellen des Eigenschaftssatzes zu nahe an der Realisierung zu orientieren. Unbewusst richtet man die Beschreibung der Eigenschaften an der Implementierung und weniger an der Spezifikation aus. Dem soll die in der vorliegenden Arbeit beschriebene Methodik vorbeugen, indem sie die Beschreibung des Sollver-

haltens des Moduls durch die Einführung einer Mappingschicht weitestgehend von der konkreten Implementierung abstrahiert. Im Idealfall erfolgt die Erstellung des Eigenschaftssatzes ohne Kenntnis der tatsächlichen Realisierung.

3.5.4 Gorai et. al., Directed-Simulation Assisted Formal Verification of Serial Protocol and Bridge

In [9] beschreibt Gorai et. al. einen ebenenbasierten Ansatz zur Verifikation serieller Protokolle. Er macht sich dabei den Umstand zu nutze, dass serielle Protokolle im Allgemeinen, wie im OSI-Modell [20] beschrieben, schichtweise definiert sind. Die unteren Schichten stellen damit Dienste bereit, auf die übergeordnete Schichten zugreifen können. Diesen Aufbau machen sich Gorai et. al. zu nutze, indem sie auch die Überprüfung der Eigenschaften an den Schichten orientieren. Wurde eine Schicht erfolgreich verifiziert, so werden diese Dienste als bewiesene Annahmen in die Eigenschaften zur Überprüfung der darüber liegenden Schicht übernommen. Auf diese Art und Weise wird das Protokoll Schicht für Schicht aufgebaut.

Die Arbeit von Gorai basiert darauf, dass das Protokoll in einzelne Layer unterteilbar ist, die sich in ihren Funktionen klar von einander abgrenzen lassen. Nur so ist es möglich, bewiesene Eigenschaften eines Layers als Annahmen in das nächste Layer zu übernehmen. Die hier vorliegende Arbeit ist auf eine solche Unterteilung des Protokoll in einzelne Layer nicht angewiesen, da sie von einer ganzheitlichen Betrachtung des Protokolls ausgeht. Trotzdem ist sie jedoch in der Lage, auch Protokolle abzubilden, deren Spezifikation in Layer unterteilt ist.

Weiterhin verwendet Gorai et. al. einen hybriden Ansatz zur Verifikation. Dabei wird nur ein Teil der Funktionalität über formale Eigenschaften nachgewiesen. Die dafür angenommenen Ausgangsbedingungen werden simulativ erreicht. Damit sind allerdings für diese simulierten Anteile keine definitiven Aussagen über die korrekte Funktion möglich.

Als Anwendungsbeispiel für seine Methodik wählte Gorai den I^2C-Bus. Dabei handelt es sich um einen verhältnismäßig einfachen seriellen Bus, bei dem es keinen inneren Zusammenhang zwischen einer Anfrage und der Antwort eines anderen Teilnehmers gibt, wie es beim Diagnosemodus des in der vorliegenden Arbeit betrachteten LIN-Busses der Fall ist. Somit kann keine Aussage getroffen werden, ob mit der Methode von Gorai ein solches Zeitverhalten verifizierbar ist.

4 Methodik zur Verifikation serieller Kommunikationsprotokolle

4.1 Schichtenbasierter Ansatz zur Verifikation von seriellen Protokollen

Im Folgenden soll ein kurzer Überblick über die in dieser Arbeit propagierte Methodik zur Anwendung formaler Verifikation auf serielle Kommunikationsprotokolle gegeben werden. Die einzelnen Teilaspekte werden danach detailliert in den jeweiligen Unterkapiteln beschrieben.

Eine der Hauptzielsetzungen dieser Arbeit ist es, die Arbeit des Verifikationsingenieurs möglichst weit unabhängig von der Implementierung und den damit befassten Personen zu machen. Damit sollen Fehler vermieden werden, die aus der wechselseitigen Kenntnis der Arbeit der anderen Teams entstehen können (jedes Team implementiert unbewusst das, was das andere Team erwartet). Herkömmliche Bottom-Up Ansätze zur formalen Verifikation testen zuerst kleine Teilmodule einer Implementierung und bauen daraus dann Schritt für Schritt komplexere Module auf. Um die einzelnen Module verifizieren zu können, müssen neben der gewünschten Funktionalität mindestens die verwendeten Ein- und Ausgangssignale für den Verifikateur bekannt sein. Erst dann kann er mit seiner Arbeit beginnen. Dies ist entweder dann der Fall, wenn die Module fertig implementiert sind und für den Test bereit gestellt werden oder sobald eine sehr detaillierte Entwurfsplanung existiert, in der die einzelnen Module mit ihren Schnittstellen und Funktionalitäten festgehalten sind. In beiden Fällen kann der Verifikations-

ingenieur erst nach diesen Vorarbeiten mit seiner Arbeit beginnen. Dadurch verlängert sich automatisch die Dauer bis zur Fertigstellung des Gesamtprojektes aus Entwicklung und Verifikation.

Weiterhin wird die Funktionalität damit auf einer Ebene geprüft, die schon sehr stark von der Interpretation der Spezifikation durch den Entwickler geprägt ist. Der Verifikateur wird damit gewissermaßen gezwungen, den Gedanken und Annahmen des Entwicklers zu folgen. Die Implementierung wird damit nur auf Einhaltung der Vorgaben des Entwicklers getestet, nicht aber auf Übereinstimmung mit der Spezifikation selbst. Es besteht die Gefahr, dass hierbei auftretende Fehlinterpretationen nicht entdeckt werden, die schon früh im Entwurfsprozess aufgetreten sind.

Aus diesem Grund basiert die hier vorgestellte Methodik im wesentlichen auf einem Top-Down Ansatz, bei dem nach Möglichkeit ausschließlich die Spezifikation als Basis für die Entwicklung und Verifikation herangezogen wird.

Wie in Abbildung 4.1 zu sehen ist, unterteilt die Methodik den Prozess dazu in drei verschiedene Schichten, die wechselseitig miteinander verbunden sind. Dies ist zum einen die Implementierungsschicht, in der die eigentliche Implementierung des Gerätes in Hardware Description Language (HDL) Code erfolgt. Auf der anderen Seite liegt die Verifikationsschicht, in der die Eigenschaftssätze zum Nachweis der Einhaltung der vorgegebenen Spezifikation erstellt werden. Ausgangspunkt für die Arbeit in beiden Schichten ist dabei immer die Spezifikation und nicht die jeweils andere Schicht. Um nun für die eigentliche Durchführung der Verifikation beide Schichten miteinander verbinden zu können, ist als dritte Schicht die Mappingschicht notwendig. Auf diese Einheit kann dann das Tool zur Formalen Verifikation angewendet werden.

Ausgangspunkt für die Methode ist in jedem Fall die Spezifikation. Es wird dabei in dieser Arbeit davon ausgegangen, dass die Spezifikation in natürlicher Sprache und nicht in formaler, maschinenlesbarer Darstellung vorliegt. Nur in diesem Fall besteht die Möglichkeit der Fehlinterpretation, die zu

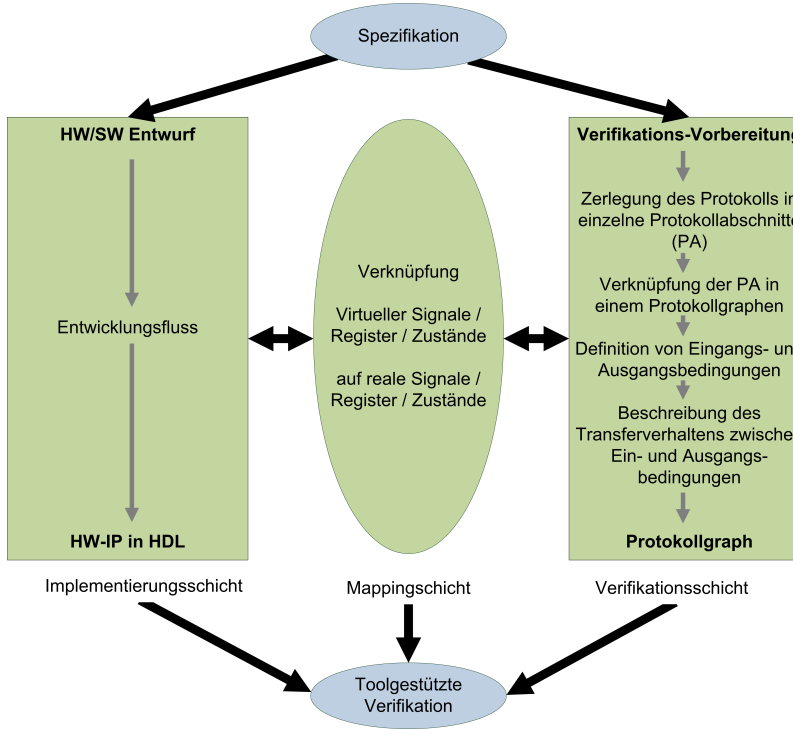


Abbildung 4.1: Verifikationsmodell

Fehlern in der Implementierung führen kann. Auf Basis der Spezifikation erstellt der Entwickler unter Verwendung bekannter Entwicklungsmethoden und -werkzeugen seine Implementierung der Spezifikation. Hierauf wird im Rahmen dieser Arbeit nicht weiter eingegangen.

Auch der Verifikateur verwendet die Spezifikation als Basis für die Erstellung der Verifikationsschicht. Eines der wichtigsten Kriterien dabei ist, dass die Formulierung der Eigenschaftsbeschreibungen möglichst unabhängig von einer konkreten Implementierung erfolgen soll. Dieses Vorgehen bietet mehrere Vorteile:

- **Erneute Interpretation der Spezifikation:** Bei der Erstellung der Verifikationsschicht muss der Verifikateur die in informeller Darstellung gegebenen Spezifikation interpretieren und in Eigenschaftsbeschreibungen umsetzen. Da diese Spezifikation in informeller Darstellung gegeben ist, enthält sie häufig Spielraum für Interpretationen, die damit wiederum Quelle von Fehlern sein können. Durch eine unabhängige Interpretation der Spezifikation durch den Entwickler und den Verifikateur können solche Interpretationsspielräume leichter aufgedeckt werden, da sich bei unterschiedlicher Interpretation die Implementierung und die Eigenschaftsbeschreibungen widersprechen.
- **Wiederverwendbarkeit für andere Implementierungen:** Gewöhnlich werden die Eigenschaftsbeschreibungen für die Verifikation einer konkreten Implementierung der Spezifikation erstellt. Dazu greifen sie auf die in der Implementierung vorhandenen Signal, Register und Zustände zurück. Außerdem wird das Ablaufverhalten häufig ebenso an der Realisierung orientiert. Damit sind diese einmal erstellten Eigenschaftsbeschreibungen aber nur für eine einzige Implementierung verwendbar. Für eine andere Implementierung müssen neue Eigenschaftsbeschreibungen erstellt werden. Wenn es nun jedoch gelingt, die Eigenschaftsbeschreibungen unabhängig von einer konkreten Implementierung zu formulieren, so können diese unverändert auch auf andere Implementierungen des gleichen Protokolls angewendet werden. Daraus

erwächst die Möglichkeit, die Gesamtmenge der Eigenschaftsbeschreibungen quasi als Konformitätsnachweis für eine Implementierung zu verwenden.

Um die Unabhängigkeit der Eigenschaftsbeschreibungen von der Implementierung zu erreichen, wird in der hier vorgestellten Methode die Verifikationsschicht durch eine zwischengeschaltete Mappingschicht von der Implementierungsschicht getrennt. Wie in Abbildung 4.2 zu sehen ist, bilden die Eigenschaftsbeschreibungen den Kern der Verifikationsschicht. Sie beschreiben, wie sich aus den am Eingang gegebenen Startbedingungen der neue Ausgangszustand des Systems ergibt. Dies wird im Folgenden als Transferverhalten des Systems bezeichnet. Nach der Verifikationsschicht folgt die Mappingschicht, die eine Anpassung an die zu verifizierende Intellectual Property (IP) vornimmt. Diese wiederum befindet sich in der Implementierungsschicht. Wie man sehen kann, existiert für jede unterschiedliche IP eine eigene Mappingschicht, die die Spezifika der Implementierung aufgreift. Die übergeordnete Verifikationsschicht bleibt jedoch für unterschiedliche IPs immer die gleiche, da die Abstraktion und/oder Anpassung in der Mappingschicht erfolgt.

Das Verhalten eines elektronischen Systems wird üblicherweise durch seine Eingangssignale sowie den Wert von speichernden Elementen wie zum Beispiel Registern bestimmt, die den aktuellen Zustand des Systems definieren. Um eine möglichst einfache Verknüpfung erreichen zu können, ist es sinnvoll, bei der Erstellung der Eigenschaftsbeschreibungen auch auf die gleiche Art von Elementen zurück zu greifen. Da diese jedoch allein in der Verifikationsschicht Verwendung finden, werden sie in dieser Arbeit als *virtuelle Elemente* bezeichnet. Sie stellen damit die Elemente einer virtuellen IP dar, die allein dem Zweck dient, die Vorgaben der Spezifikation in Eigenschaftsbeschreibungen umzusetzen.

Um nun eine konkrete Implementierung mit diesem Eigenschaftssatz überprüfen zu können ist es notwendig, die virtuellen Elemente der Eigenschaftsbeschreibungen mit den realen Elementen der Implementierung zu verknüp-

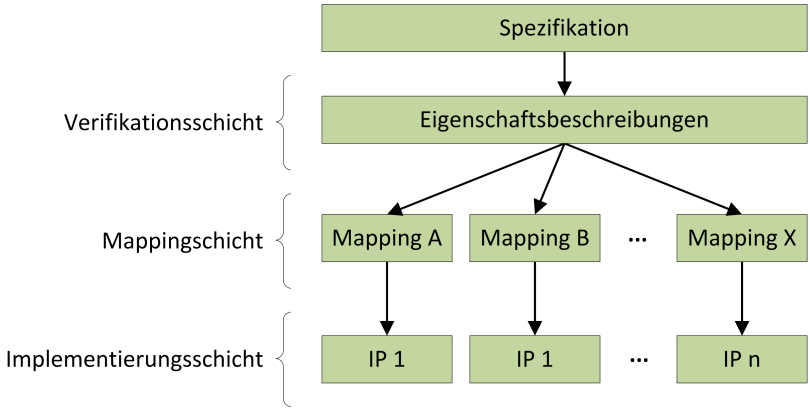


Abbildung 4.2: Prinzip der Mappingschicht

fen. Dies ist die Aufgabe der Mappingschicht. Diese Schicht ist damit nicht mehr universell, sondern spezifisch für die zu überprüfende Implementierung. Da zur Erstellung dieser Schicht sowohl Informationen zum Verifikationsmodell als auch zur Implementierung benötigt werden, muss diese Schicht möglicherweise in gemeinsamer Arbeit durch den Entwickler und den Verifikateur erstellt werden.

Das grundsätzliche Vorgehen zur Erstellung der Verifikationsschicht und der Mappingschicht ist also wie folgt:

1. Analyse des Protokolls und Zerlegung in einzelne Protokollabschnitte (siehe Kapitel 4.4)
2. Beschreibung der einzelnen Protokollabschnitte (siehe Abschnitt 4.5)
   a) Definition der Ein- und Ausgangsbedingungen
   b) Definition des Transferverhaltens
3. Erstellen der Mappingschicht (siehe Abschnitt 4.6)

4.2 Definitionen und Terminologie

Im diesem Abschnitt werden einige für das Verständnis der Arbeit nötige Begriffe definiert.

4.2.1 Verifikationsschicht

Definition 4.1 (Protokollgraph)**.** Graphische Darstellung des Protokolls. Dabei repräsentieren die Knoten eine Sequenz beliebiger Länge aus dem Protokoll. Die Kanten geben die logische Verknüpfung dieser Knoten an.

Definition 4.2 (Protokollabschnitt)**.** Ein Knoten des Protokollgraphen. Er repräsentiert eine beliebig lange Sequenz aus dem Protokoll.

Definition 4.3 (Protokollsequenz)**.** Ein Pfad durch den Protokollgraphen. Dabei werden mehrere Protokollabschnitte durchlaufen.

Definition 4.4 (Eingangsbedingung)**.** Menge an Zuständen und Signalbelegungen, die den Beginn eines Protokollabschnitts definieren.

Definition 4.5 (Ausgangsbedingung)**.** Menge an Zuständen und Signalbelegungen, die das Ende eines Protokollabschnitts definieren

Definition 4.6 (Virtuelles Register)**.** Konzeptionelles Register zur abstrakten Beschreibung eines Protokollablaufes. Es muss keine direkte Entsprechung zu einem realen Register geben.

Definition 4.7 (Virtuelles Signal)**.** Konzeptionelles Signal zur abstrakten Beschreibung eines Protokollablaufes. Es muss keine direkte Entsprechung zu einem realen Signal geben.

Definition 4.8 (Virtueller Zustand)**.** Konzeptioneller Zustand zur abstrakten Beschreibung eines Protokollablaufes. Es muss keine direkte Entsprechung zu einem realen Zustand geben.

4.2.2 Mappingschicht

Definition 4.9 (Verknüpfungsmatrix)**.** Gibt die Verknüpfung der virtuellen Register/Signale/Zustände mit realen Registern/Signalen/Zuständen im zu verifizierenden IP-Core wieder. Einem virtuellen Register/Signal/Zustand können dabei mehrere reale Register/Signale/Zustände zugeordnet werden.

4.2.3 Implementierungsschicht

Definition 4.10 (Reales Register)**.** In der Implementierung des IP-Cores vorhandenes Register.

Definition 4.11 (Reales Signal)**.** In der Implementierung des IP-Cores vorhandenes Signal.

Definition 4.12 (Realer Zustand)**.** In der Implementierung des IP-Cores vorhandener Zustand einer Finite State Machine (FSM).

4.3 Beispielprotokoll zur Darstellung der Methodik

Um die in dieser Arbeit vorgestellte Methodik etwas anschaulicher darstellen zu können, soll hier kurz ein fiktives Beispiel für ein einfaches serielles Protokoll beschrieben werden. Anhand dieses Protokolls werden dann im weiteren Verlauf die einzelnen Schritte der Methodik beschrieben.

Um das Protokoll möglichst einfach zu halten, wird von einem System mit einem zentralen Master und mehreren angeschlossenen Slaves ausgegangen. Aus diesen Grund ist keine Arbitrierung notwendig. Die kleinste Übertragungseinheit bei diesem Protokoll ist ein Byte, bestehend aus acht Bits. Da es sich um ein asynchrones Protokoll ohne zusätzliche Taktleitung handelt, wird jedes Byte von einem dominanten Startbit und mindestens einem Stopbit eingerahmt (siehe Abbildung 4.3).

Abbildung 4.3: Darstellung eines Bytes im Beispielprotokoll

Aus diesen einzelnen Bytes kann dann das Rahmenformat zusammengesetzt werden. Ein Datenrahmen besteht aus einem ein Byte langen Header mit Steuerinformationen, 0-3 Datenbytes und einem Byte Trailer, in dem Informationen zur Fehlersicherung enthalten sind (z.B. CRC). Einen Überblick über das Rahmenformat des Beispielprotokolls gibt Abbildung 4.4.

Abbildung 4.4: Rahmenformat des Beispielprotokolls

Im Header befindet sich an erster Stelle die 5 Bit lange Adresse des angesprochenen Slaves. Mit Hilfe des nächsten Bits wird zwischen einem Lese- und einem Schreibzugriff unterschieden. Die folgenden zwei Bits geben die Anzahl der nach dem Header folgenden Datenbytes an. Diese bilden anschließen den Mittelteil des Datenrahmens. Den Abschluss des Rahmens bildet eine ein Byte lange Checksumme zur Fehlererkennung bei der Übertragung.

4.4 Erstellung des Protokollgraphen zur Abbildung des Protokollablaufes

In diesem Abschnitt soll das Vorgehen zur Erstellung des Protokollgraphen aus der Spezifikation beschrieben werden. Dazu muss in einem ersten Schritt die Beschreibung in der Spezifikation vom Verifikateur untersucht und in sinnvolle Protokollabschnitte aufgeteilt werden. Die Abschnitte können sowohl linear aufeinander folgen als auch Verzweigungspunkte bilden. Im Anschluss daran müssen die einzelnen Protokollabschnitte zum Protokollgra-

phen verknüpft werden, so dass jeder Protokollabschnitt mindestens einen Vorgänger und Nachfolger zugewiesen bekommt. In seiner Gesamtheit beschreibt dieser das vollständige Protokollverhalten. Um den sauberen Anschluss der einzelnen Protokollabschnitte zu garantieren, müssen die Ein- und Ausgangsbedingungen jeweils exakt zueinander passen. Verzweigungen werden durch unterschiedliche Ausgangsbedingungen der Eigenschaftsbeschreibung abgebildet. Sie bestimmen den weiteren Pfad im Protokollgraphen.

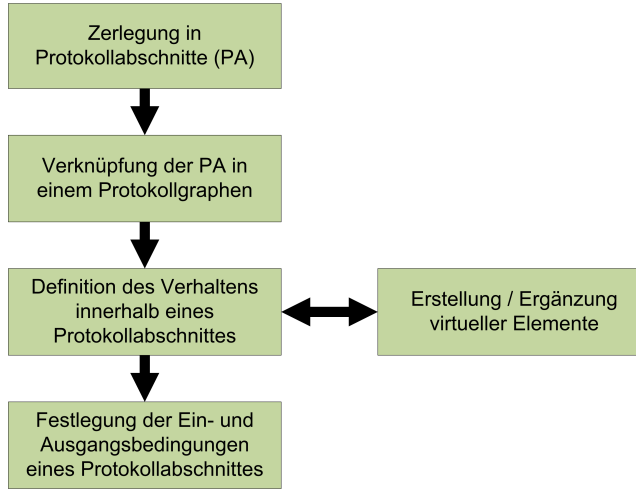


Abbildung 4.5: Stufenweise Erstellung des Protokollgraphen

4.4.1 Protokollzerlegung

Grundlage für die Abbildung des durch ein Protokoll definierten Verhaltens in einen Protokollgraphen ist die Zerlegung des Protokolls in sinnvolle Protokollabschnitte. Aus der Summe des Verhaltens dieser einzelnen Protokollabschnitte entsteht dann die Beschreibung des Gesamtverhaltens des Pro-

tokolls. Eine automatisierte Bestimmung dieser Protokollabschnitte aus der natürlichsprachigen Darstellung der Spezifikation ist eine komplexe Aufgabe, die nicht Inhalt der vorliegenden Arbeit ist. Es handelt sich hier also um einen Vorgang, der vom Verifikateur manuell durchgeführt werden muss. Dabei ist er zu einem nicht unerheblichen Teil auf sein Wissen und seine Erfahrung angewiesen. Aufgrund der großen Bandbreite möglicher Protokolle können an dieser Stelle auch keine allgemeingültigen Vorgaben für ein „richtiges“ Vorgehen gemacht werden. Statt dessen sollen hier Kriterien und Überlegungen aufgezeigt werden, die den Verifikateur bei seiner Arbeit unterstützen können.

Ziel des Prozesses ist es, das gesamte Verhalten des Protokolls in einzelne, in sich abgeschlossenen Abschnitte zu unterteilen, deren gewünschtes Verhalten im Anschluss in Form einer Eigenschaftsbeschreibung festgelegt werden kann. Eine grobe Struktur, die jeweils große Teile der Funktionalität in einer Eigenschaftsbeschreibung zusammenfasst, ermöglicht eine übersichtliche und schnell zu erfassende Darstellung des Protokolls im Protokollgraphen. Dem gegenüber steht jedoch, dass so das zu betrachtende Zeitfenster für die formale Verifikation sehr groß werden kann. Damit steigt aber direkt die Komplexität und der Aufwand für die Überprüfung der Eigenschaft. Der Aufwand kann sogar so groß werden, dass er mit aktuell verfügbaren Werkzeugen nicht mehr handhabbar ist. Umgekehrt führt eine Zerlegung des Verhaltens in sehr kleine Abschnitte zu kompakten Eigenschaftsbeschreibungen mit kleinen Beobachtungsfenstern. Diese lassen sich einfach und übersichtlich beschreiben, da dort jeweils nur eine kurzer Zeitraum beobachtet werden muss. Dem steht jedoch eine sehr unübersichtliche Darstellung des Protokolls gegenüber. Zudem sollte die Darstellung des Protokolls möglichst allgemein und abstrakt gehalten bleiben, um nicht schon im Vorfeld die Beschreibung in Richtung einer möglichen Hardwarestruktur zu erstellen.

Aus der Untersuchung verschiedener Kommunikationsprotokolle (z.B. LIN [14], CAN, I^2C, ASI) bietet sich hier eine Unterteilung in die auch schon in vielen Protokollen definierten Grenzen. Dabei handelt es sich im Allgemei-

nen um eine Unterteilung in Bytes oder Wörter. Diese bilden die Felder aus denen der Datenrahmen der Protokolle zusammen gesetzt ist. Vielfach werden die Steuerinformationen und Daten in Vielfachen dieser Einheiten übertragen und bearbeitet.

Ein Kriterium für die Zerlegung des Protokolls in einzelne Abschnitte sind mögliche Entscheidungspunkte, an denen sich das Verhalten der Implementierung in Abhängigkeit vom aktuellen Zustand und den Eingaben ändern kann. Dies könnte zum Beispiel die Unterscheidung von Lese- und Schreibzugriffen oder der Wechseln von der Datenübertragung zur Verarbeitung von Steuerinformationen sein. Grundsätzlich sollte eine solche Entscheidungsstelle in mindestens drei Protokollabschnitte unterteilt werden. Der Zustand des Systems vor der Entscheidung wird durch einen eigenen Protokollabschnitt festgelegt. Für jede mögliche Fallunterscheidung sollte dann ein weiterer Protokollabschnitt vorgesehen werden. In diesen Protokollabschnitten muss dann das Kriterium festgelegt werden, nachdem einer der verschiedenen möglichen Protokollabschnitte für den weiteren Verlauf ausgewählt wird. Weitere Details zur Beschreibung dieser Entscheidungskriterien finden sich im Abschnitt 4.5.

Bei dem in Kapitel 4.3 vorgestellten Beispiel biete sich eine Trennung an den Grenzen der einzelnen Bytes an. Diese bilden jeweils eine abgeschlossene Einheit, wie zum Beispiel den Header oder die einzelnen Datenbytes. Eine feinere Untergliederung würde die Darstellung des Protokolls als Eigenschaftssatz erschweren, da der im Protokoll implizit schon vorhandene Zusammenhang zwischen einzelnen Bits aufgebrochen würde. So befinden sich alle für den weiteren Verlauf der Übertragung benötigten Informationen im Header. Auch die Daten selbst werden in Blöcken zu jeweils 8 Bit weiterverarbeitet.

Auf der anderen Seite erscheint es wenig sinnvoll, größere Einheiten zu bilden. Der Header bildet eine in sich geschlossene Einheit, da die dort vorhandenen Informationen den weiteren Verlauf bestimmen. Eine Zusammenlegung mit den folgenden Daten würde bedeuten, dass für jede mögliche Grö-

ße des Rahmens eine eigene Eigenschaftsbeschreibung zu erstellen ist. Dies wäre zwar möglich, allerdings auch ineffizient wegen der großen Anzahl an benötigten Eigenschaftsbeschreibungen, die sich nur in der Anzahl der überprüften Datenbytes unterscheiden. Statt dessen bietet es sich an, das Ende des Header als Verzweigungspunkt zu nutzen und dort neue Eigenschaftsbeschreibungen beginnen zu lassen. Diese würden dann die Übertragung der Datenbytes jeweils getrennt für einen Lese- und Schreibzugriff abbilden. Den Abschluss wiederum bildet als letzter Protokollabschnitt die Fehlersicherung im Trailer. Auch hier sollte die Beschreibung wieder getrennt von den Datenbytes erfolgen.

In der Theorie sollte das Verhaltens eines Systems für alle möglichen Zustände und Eingänge in der Spezifikation festgelegt sein. In der Realität ist die Vollständigkeit bei der Spezifikation jedoch nur in den seltensten Fällen gegeben. Damit ergeben sich also Lücken bei der Beschreibung des Protokollgraphen. Außerdem wurde schon im Kapitel 3.1 festgelegt, dass in dieser Arbeit nur der Nachweis von protokollkonformem Verhalten, nicht aber die Untersuchung auf Robustheit der Implementierung betrachtet werden soll. Aus diesem Grund wird es im Protokollgraph Stellen geben, an denen das Verhalten des Systems nicht näher beschrieben wird. Diese Protokollabschnitte stellen damit verbotene oder undefinierte Zustände dar, für die das Verhaltens des Systems nicht festgelegt ist. Diese Stub-Properties sollten trotzdem angelegt werden, um alle Entscheidungsmöglichkeiten mit einer Reaktion zu belegen. Die Stub-Properties, die sich aus einer Unvollständigkeit der Spezifikation ergeben, können im Anschluss eventuell genutzt werden, um die Spezifikation in dieser Hinsicht zu überarbeiten.

4.4.2 Verknüpfung der Protokollabschnitte zum Protokollgraphen

Die bisher definierten Protokollabschnitte stehen noch losgelöst voneinander. Ein zusammenhängender Ablauf im Sinne der Spezifikation ergibt sich jedoch erst durch die Verkettung. Dazu wird für jeden einzelnen Protokoll-

abschnitt festgelegt, welche Vorgänger und Nachfolger er besitzt. Mehrere Nachfolger treten dann auf, wenn ein Protokollabschnitt eine Entscheidungsstelle des Protokolls beschreibt. Wie in Abschnitt 4.5 noch näher beschrieben wird, überprüft jeder Folgeabschnitt an einer Entscheidungsstelle genau eine der Entscheidungsmöglichkeiten. Ähnlich des Schlüssel-Schloss-Prinzips passt dieser Protokollabschnitt also nur auf eine mögliche Kombination der virtuelle Elemente. Unter Kombination kann dabei allerdings auch eine Menge von Kombinationen der virtuellen Elemente gemeint sein (z.B. erlaubte PIDs).

Umgekehrt ist es natürlich genauso möglich, dass ein Zustand verschiedene Protokollabschnitte als Vorgänger hat. Das ist notwendig, um die verschiedenen Protokollabschnitte nach einer Entscheidungsstelle auch wieder zusammen zu führen. Ein klassisches Beispiel dafür ist der Ruhezustand, in den das Protokoll nach einer erfolgreichen Transaktion zurück führt. Im Gegensatz zur Entscheidungsstelle müssen hier allerdings keine Bedingungen überprüft werden. Der eine Protokollabschnitt folgt einfach auf den anderen.

Bei Betrachtung des Beispiels in Kapitel 4.3 ergibt sich der in Abbildung 4.6 gezeigte Protokollgraph.

4.5 Beschreibung des Transferverhaltens

Nach der Zerlegung des Protokolls in die einzelnen Protokollabschnitte und der Verkettung dieser Abschnitte zu einem Protokollgraphen ist die grundlegende Struktur und der Verlauf des Protokolls abgebildet. Im nächsten Schritt muss nun die eigentliche Funktionalität abgebildet werden. Die Beschreibung des gewünschten Verhaltens erfolgt dabei in Form von Eigenschaftsbeschreibungen in der Sprache des gewählten Tools zur formellen Verifikation. Pro Protokollabschnitt entsteht in diesem Schritt eine Eigenschaftsbeschreibung, die das gewünschte Verhalten dieses Abschnittes abbildet. Somit entsteht iterativ aus den einzelnen Eigenschaftsbeschreibun-

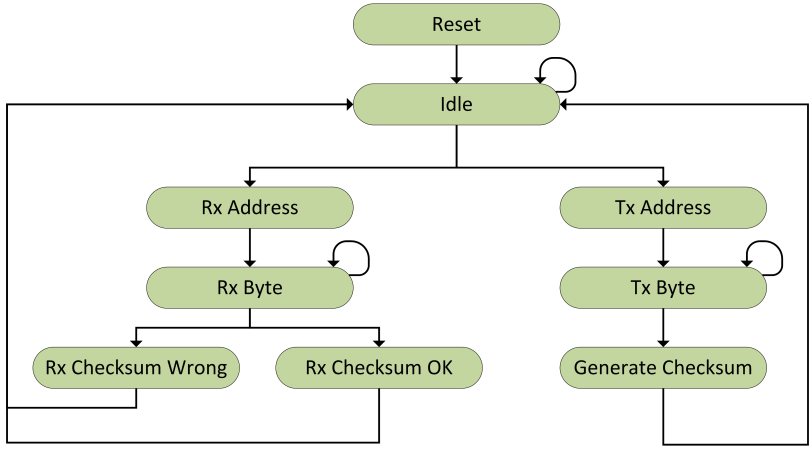


Abbildung 4.6: Protokollgraph des Beispielprotokolls

gen der Protokollabschnitte die gesamte Beschreibung des zu verifizierenden Protokolls.

Zur Erläuterung sei auf Abbildung 4.7 verwiesen. Hier sind zwei Eigenschaftsbeschreibung (A und B) zu sehen, die aufeinander folgen sollen und so zu einem Protokollgraphen zusammengesetzt werden. Damit dies möglich ist, müssen sie zwingend einige Kriterien erfüllen. So ist eine gemeinsame Beschreibung der Ein- und Ausgänge des Systems sowie des internen Zustands notwendig. Diese Beschreibung muss sich auf konkrete Implementierungen abbilden lassen, ohne diese direkt nachzubilden. Dazu werden in folgenden Kapitel die virtuellen Elemente eingeführt. Bei ihnen handelt es sich um Signale, Register und Zustände, die für die Beschreibung des gewünschten Systemverhaltens benötigt werden. Sie können, müssen aber nicht realen Signalen, Registern und Zuständen in einer möglichen Implementierung entsprechen.

Damit die verschiedenen Protokollabschnitte für die formale Verifikation sauber aneinander passen, ist es erforderlich, dass die Annahmen zu Beginn einer Eigenschaftsbeschreibung innerhalb des vorhergehenden Protokollabschnittes bewiesen werden. Daraus folgt die Definition der so genannten Eingangs- und Ausgangsbedingungen und deren korrekte Verknüpfung. Der zentrale Punkt zur Abbildung der Spezifikation ist dann die Beschreibung des Überführungsverhaltens von der Eingangs- zur Ausgangsbedingung durch die Formulierung des Beweisteils für die formale Verifikation. Hiermit wird die korrekte Reaktion des Systems auf Eingangssignale und den internen Zustand festgelegt.

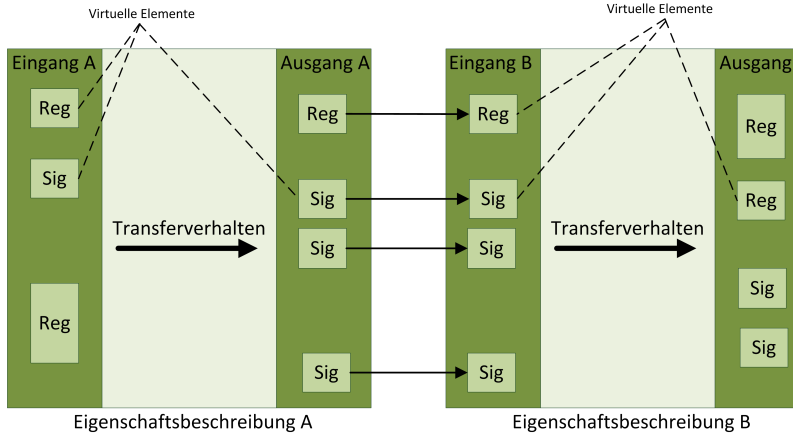


Abbildung 4.7: Transferverhalten

4.5.1 Virtuelle Elemente zur Abstraktion von einer konkreten Implementierung

Das Ziel bei der Abbildung des Protokolls in einen Eigenschaftssatz ist, wie schon in Kapitel 3.1 angesprochen, die Unabhängigkeit der Beschreibung

von einer konkreten Implementierung. Nichts desto trotz soll sich der erstellte Eigenschaftssatz später auf eine konkrete Realisierung anwenden lassen, um diese auf Konformität zur Spezifikation hin zu überprüfen. Aus diesem Grund erscheint es angeraten, sich für die Beschreibung der Funktionalität auf Strukturen zu stützen, wie sie üblicherweise auch in realer Hardware vorkommen. Zu diesem Zweck werden in dieser Arbeit so genannte virtuelle Signale, Register und Zustände eingeführt. Mit deren Hilfe kann das Verhalten in einer Art und Weise beschrieben werden, die beim späteren Mapping eine einfache Zuordnung zur den realen Signalen, Registern und Zuständen ermöglicht.

Ein virtuelles Signal sollte zum Beispiel verwendet werden, um die Kommunikation mit der Außenwelt darzustellen. Informationen, die über die externen Schnittstellen des Gerätes übernommen oder verschickt werden, können im Allgemeinen als virtuelles Signal dargestellt werden. Dazu gehören auf jeden Fall die Signale, die den Bus bilden. Zusätzlich sollten weitere Signale vorgesehen werden, die der Kommunikation mit der Hostseite dienen, an die die Daten vom Bus übergeben, beziehungsweise zu sendende Daten übernommen werden. Ähnlich verhält es sich bei Informationen, die innerhalb des Protokollgraphen gespeichert werden müssen, auch wenn sich die auslösenden Signale verändert haben. Für diesen Zweck werden die virtuellen Register eingesetzt. Da Protokolle bei der Spezifikation und bei der späteren Realisierung oft in Form von FSM dargestellt werden, stehen auch virtuelle Zustände für die Abbildung der Funktionalität der Spezifikation in Eigenschaftsbeschreibungen zur Verfügung.

Bei der Bestimmung der benötigten virtuellen Elemente handelt es sich im Allgemeinen um einen iterativen Prozess. Die Signale des Busses und für die Kommunikation mit dem Host lassen sich schon zu Beginn der Beschreibung festlegen. Ähnlich ist es mit den wichtigsten virtuellen Zuständen, die sich oftmals direkt aus der Protokollspezifikation ergeben. Die Erfahrung hat jedoch gezeigt, dass in diesem ersten Schritt selten alle benötigten virtuellen Elemente bestimmt werden können. Im Rahmen der Erweiterung des Eigen-

schaftensatzes um weitere Protokollabschnitte muss die Menge der virtuellen Elemente also iterativ erweitert werden.

4.5.2 Definition von Eingangs- und Ausgangsbedingungen

Wie im Kapitel 2.5 beschrieben, besteht eine Eigenschaftsbeschreibung immer aus der Angabe eines Initialzustands (entspricht der Eingangsbedingung) und einem Beweisteil. In diesem Beweisteil wird der Wert der virtuellen Elemente für die Ausgangsbedingung festgelegt. Damit nun der Protokollgraph sauber aufgebaut werden kann, ist es notwendig, dass die Eingangsbedingung eines Protokollabschnittes im vorhergehenden Protokollabschnitt bewiesen und damit festgelegt wird. Nur so kann die kontinuierlich Fortführung des Systemzustandes gewährleistet werden. Die Angabe von Annahmen zum Wert einzelner virtueller Elemente zu Beginn einer Eigenschaftsbeschreibung schränkt den möglichen Ereignisraum für die formale Verifikation ein und soll sogenannte „false negatives“ verhindern. Dabei handelt es sich um Gegenbeispiele die entstehen, wenn das Verifikationstool von im realen Betrieb nicht erreichbaren Ausgangsbedingungen ausgeht. Damit ist es aber zwingend erforderlich, dass das Erreichen dieser Eingangsbedingungen an anderer Stelle auch nachgewiesen wird und nicht einfach nur auf einer intuitiven Annahme des Verifikateurs beruht. An diesem ist es jedoch nicht notwendig und auch gar nicht erwünscht, alle virtuellen Elemente in allen Protokollabschnitten genau festzulegen. Hier sollten nur diejenigen virtuellen Elemente eingebunden werden, die im aktuellen Protokollabschnitt auch eine Bedeutung haben und das Verhalten des Protokolls in diesem Abschnitt beeinflussen können. Alle anderen virtuellen Elemente sollten hingegen nicht in ihren Werten festgelegt werden. Dadurch wird erreicht, dass diese so allgemein wie möglich gehalten wird und wirklich nur unbedingt notwendige Festlegungen getroffen werden. Werden an dieser Stelle Eigenschaften festgelegt, die eigentlich in der Spezifikation offen gelassen wurden, so kann dies beim späteren Zusammenbringen von Verifikationsschicht und Implementierung zu unnötigen Gegenbeispielen führen.

Diese entstehen, weil der Entwickler die Freiheiten der Spezifikation anders genutzt hat, als durch die zu strenge Beschreibung im Protokollgraphen vorgegeben.

An dieser Stelle ist es wichtig darauf hinzuweisen, dass mit den weiter oben angesprochenen Ein- und Ausgangsbedingungen nur die (internen) virtuellen Elemente gemeint sind. Im Gegensatz dazu stehen externe Eingangssignale. Diese beeinflussen auch das Verhalten eines Protokollabschnittes, sie lassen sich jedoch aufgrund ihrer Natur nicht beweisen. Damit ergibt sich formale eine Zweiteilung der Ein- und Ausgangsbedingungen in interne virtuelle Elemente und externe Signale. Die Forderung, dass die Ein- und Ausgangsbedingungen bei zwei aufeinander folgenden Protokollabschnitten gleich sein müssen, bezieht sich nur auf die internen virtuellen Elemente. Trotzdem bilden externe Signale einen integralen Bestandteil der Ein- und Ausgangsbedingungen. Als Teil der Eingangsbedingungen legen sie die korrekte Reaktion des Protokolls auf Stimuli von Außen fest. Umgekehrt bestimmen sie als Teil der Ausgangsbedingungen das korrekte Verhalten in der Außensicht, zum Beispiel auf den Busleitungen.

4.5.3 Darstellung des Transferverhalten für Eingangs- und Ausgangsbedingung

Nach der Festlegung der Ein- und Ausgangsbedingungen kann nun das konkrete Überführungsverhalten für den Protokollabschnitt in Form einer Eigenschaftsbeschreibung festgelegt werden. Analog zu den Ausgangsbedingungen lässt sich auch diese Überführungsfunktion in mehrere Teile untergliedern. Ein Teil der Überführungsfunktion stellt sicher, dass die internen virtuellen Elemente die korrekten Werte annehmen, um den sauberen Anschluss an den folgenden Protokollabschnitt zu ermöglichen. Dabei ist zu beachten, dass nicht zwingend die gleichen virtuellen Elemente Teil der Ausgangsbedingung werden, die schon in der Eingangsbedingung vorhanden waren und umgekehrt. Der Wert eines Zählers, der die Anzahl der noch zu

übertragenden Bytes angibt, ist zum Beispiel nach der erfolgreichen Übertragung nicht mehr von Interesse. Umgekehrt können neue virtuelle Elemente wichtig werden, die bisher keine Rolle gespielt haben.

Bei Betrachtung des Beispielprotokolls aus Kapitel 4.3 trifft dies zum Beispiel für die Information zur Anzahl der im Rahmen vorhandenen Datenbytes zu. Das Prinzip dazu soll Abbildung 4.8 verdeutlichen. Die Information zur Anzahl der Bytes in dem zu empfangenden Datenpaket spielt zu Beginn keine Rolle. Deswegen ist diese Information auch nicht Teil der Eigenschaftsbeschreibung `Rx_Address`, die das korrekte Empfangen des Headers überprüft. Allerdings wird diese Information von der nachfolgenden Eigenschaftsbeschreibung `Rx_Byte` benötigt, die den korrekten Empfang der einzelnen Datenbytes prüft. Aus diesem Grund ist das virtuelle Element `Byte Count` Teil der Ausgangsbedingung der Eigenschaftsbeschreibung `Rx_Address`. Die darauf folgende Eigenschaftsbeschreibung `Rx_Byte` überprüft nun, ob bei jedem empfangenen Datenbyte der Zähler korrekt um "´1“ erhöht wird. Da die Eigenschaftsbeschreibung wiederum auf sich selbst folgen kann, ist das virtuelle Element `Byte Count` auch wieder Teil ihrer Ausgangsbedingung. Diese wird dann wiederum von der abschließenden Eigenschaftsbeschreibung `Rx Checksum OK` übernommen, die den Empfang einer korrekten Checksumme überprüft (der zweite Fall einer inkorrekten Checksumme wird hier nicht betrachtet).

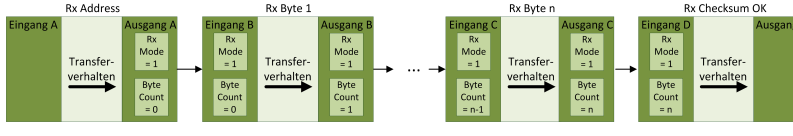


Abbildung 4.8: Ein- und Ausgangsbedingungen beim Beispielprotokoll

Eine Ausnahme von dieser Regel stellen virtuelle Elemente da, die für die Abbildung des korrekten Ablaufes bei Split-Übertragungen zum Einsatz kommen. Damit sind Übertragungen gemeint, bei denen zwischen Beginn und Ende einer Übertragung unter Umständen beliebig lange Pausen auftreten

können. In diesen Pausen können dann theoretisch beliebige weitere Übertragungen stattfinden. Dieser Fall wird in Kapitel 4.8 näher betrachtet.

Neben diesem eher linearen Verhalten können innerhalb eines Protokollabschnittes aber auch Entscheidungen getroffen werden. Ein Beispiel dafür ist die Unterscheidung zwischen verschiedenen Übertragungsmodi oder Pakettypen. Hier existieren also Stellen im Protokoll bei denen nach einem Protokollabschnitt zwei oder mehr alternative Abschnitte folgen können. Im Falle des Beispielprotokolls aus Kapitel 4.3 ist eine solche Fallunterscheidung für die Modi Lesen und Schreiben notwendig. Während der Kopf des Rahmens in beiden Fällen gleich behandelt wird, müssen im Falle eines Lesezugriff Daten vom angesprochenen Gerät empfangen werden, während beim Schreibzugriff Daten gesendet werden. Daraus ergibt sich ein klar unterschiedliches Verhalten, das am sinnvollsten in zwei getrennten Zweigen von Eigenschaftsbeschreibungen abgebildet werden sollte.

Zur Abbildung von Fallunterscheidungen innerhalb eines Protokolls sind mindestens vier Protokollabschnitte notwendig. Das Prinzip soll wieder an Hand des Beispielprotokolls dargestellt werden (siehe Abbildung 4.9). Ausgangspunkt ist ein gemeinsamer Protokollabschnitt, der vor der Entscheidungsstelle liegt. In diesem Beispiel ist das der Protokollabschnitt `Idle`. Es handelt sich hier um den Zustand in dem sich das System befindet, wenn keine Daten empfangen oder gesendet werden. Zu einer Fallunterscheidung muss es dann kommen, wenn eine Übertragung stattfinden soll. Es kann sich dabei entweder um den Empfange (Rx) oder das Senden (Tx) von Daten handeln. Diese zwei unterschiedlichen Fälle werden von den beiden folgenden Protokollabschnitten `Rx Address` oder `Tx Address` abgefangen. Beide Protokollabschnitte müssen die gleichen Eingangsbedingungen haben, da sie ja auf den gemeinsamen Vorgänger `Idle` folgen sollen.

Die eigentliche Fallunterscheidung findet in den beiden Protokollabschnitten `Rx Address` und `TX Address` statt. Dazu existieren zusätzliche Annahmen `Assume Rx` und `Assume Tx`, die die Eingangsbedingungen ergänzen. Diese Annahmen betreffen zum Beispiel den Wert des Header-Bits, mit dem

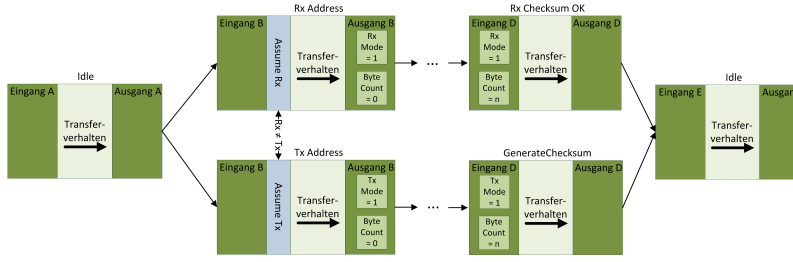


Abbildung 4.9: Fallunterscheidung beim Beispielprotokoll

zwischen Senden und Empfangen umgeschaltet wird. Dabei ist zu beachten, dass die angenommenen Werte der relevanten virtuellen Elemente der Eingangsbedingungen so gewählt werden, dass eindeutig für jede mögliche Entscheidung nur ein Protokollabschnitt zugeordnet wird. Es darf also keine Doppeldeutigkeiten geben. Dies entspricht bildlich gesprochen dem Schlüssel-Schloss-Prinzip, mit dem jeder möglichen Fallunterscheidung (Schlüssel) genau ein Protokollabschnitt (Schloss) zugeordnet wird. Umgekehrt ist jedoch auch darauf zu achten, dass für jeden möglichen Fall ein eigener dazu passender Protokollabschnitt existiert. Die Menge der abgedeckten Protokollabschnitte muss also vollständig sein. Nur so kann sichergestellt werden, dass das Verhalten der späteren Realisierung für alle möglichen Fälle abgedeckt ist. Hierbei müssen unter Umständen die schon im Kapitel 4.4.1 angesprochenen Stub-Properties eingesetzt werden. Es ist also auch möglich, dass eine Fallunterscheidung in mehr als zwei Fälle getroffen wird.

Es ist nicht ungewöhnlich, dass in der Spezifikation bei einer Fallunterscheidung nicht alle möglichen Fälle erfasst und mit einem gewünschten Verhalten beschrieben sind. Dies ist häufig der Fall, wenn kein konkretes Verhalten für den Fehlerfall festgelegt ist. Die Spezifikation geht nur von einer fehlerfreien Abarbeitung des Protokolls aus und macht keine Angaben dazu, was passieren soll, wenn die empfangenen Informationen nicht den Vorgaben des Protokolls entsprechen. Um nun trotzdem eine vollständige Abdeckung

des möglichen Verhaltens des Systems zu erreichen, müssen an diesen undefinierten Stellen so genannte Stub-Properties eingefügt werden. Bei diesen stimmen, wie bei den anderen Protokollabschnitten, die zu einer Fallunterscheidung gehören, die Eingangsbedingungen mit den Ausgangsbedingungen des vorherigen Protokollabschnittes überein. In den für die Entscheidung relevanten Annahmen finden sich hier die Fälle, die bei den übrigen Protokollabschnitten nicht abgedeckt wurden. Im Gegensatz zu diesen ist jedoch kein Verhalten in der Eigenschaftsbeschreibung abgebildet. Außerdem verfügen sie nicht über einen nachfolgenden Protokollabschnitt. Sie stellen damit eine Sackgasse dar, von der aus es kein korrektes weiteres Verhalten gibt. Dies mag auf den ersten Blick gravierend erscheinen, da damit die Kette der Verifikation unterbrochen ist und kein Verhalten des Systems definiert wird. Allerdings können diese Stub-Properties nur unter zwei Bedingungen eingenommen werden. Zum einen dann, wenn die Spezifikation nicht vollständig oder fehlerhaft ist. Unter diesen Umständen ist aber kein Nachweis korrekten Protokollverhaltens möglich. Hierzu muss also erst die Spezifikation entsprechend angepasst werden, die Methodik gibt hier also einen Hinweis auf mögliche Fehler in der Spezifikation. Zum anderen könnte einer der in den Stub-Properties beschrieben Zustände dann eingenommen werden, wenn das System fehlerhafte Eingangssignale empfängt und die Spezifikation für diesen Fall keine Vorgaben zum weiteren Vorgehen macht. Dies ist allerdings Teil einer Robustheitsuntersuchung, die für diese Arbeit explizit ausgeschlossen wurde.

Innerhalb der durch die Fallunterscheidung vorhandenen Protokollabschnitte wird nun das weitere gewünschte Verhalten für den dadurch abgedeckten Fall beschrieben. Für das Beispielprotokoll wäre da also das korrekte Empfangen oder Senden der folgenden Datenbytes. Dabei kann für eine Fallunterscheidung jeweils nur ein einzelner Protokollabschnitt als Folge oder aber auch eine weitere Kette von Protokollabschnitten auftreten. Eine Möglichkeit, wiederkehrendes Verhalten in einem einzigen Protokollabschnitt abzubilden wird später in diesem Kapitel vorgestellt werden. Innerhalb dieser Ketten von Protokollabschnitten sind auch wieder weitere Fallunterschei-

dungen möglich. Für die quasi parallelen Ketten der Protokollabschnitte, die sich aus der Fallunterscheidung ergeben, gilt, dass sich die Eingangs- und Ausgangsbedingungen der jeweiligen Protokollabschnitte voneinander unterscheiden. Eine Gleichheit würde bedeuten, dass die Protokollabschnitte untereinander ausgetauscht werden könnten und damit die Fallunterscheidung aufgehoben ist.

Den Abschluss der Fallunterscheidung bildet ein Protokollabschnitt, in dem die verschiedenen Ketten wieder zusammengeführt werden. Für die hier betrachteten seriellen Protokolle ist das im Allgemeinen der Ruhezustand, der nach der Datenübertragung eingenommen wird. Aber auch innerhalb einer laufenden Übertragung sind solche zusammenfassenden Protokollabschnitte denkbar. Ein Beispiel dafür könnte die Berechnung einer Checksumme sein, die den Abschluss verschiedener Lesemodi bildet. Während sich die Behandlung der Daten im Vorfeld unterscheidet, wird die Chechsumme wieder gleich berechnet. In jedem dieser Fälle ist es so, dass auch hier die Eingangsbedingungen des Protokollabschnittes den Ausgangsbedingungen der vorhergehenden Protokollabschnitte entsprechen müssen. Im Allgemeinen gibt es dazu nur wenige virtuelle Elemente, die in diesen Bedingungen erfasst werden müssen, da die Entscheidungen schon in den vorhergehenden Protokollabschnitten abgearbeitet wurden.

Bei der seriellen Datenübertragung kommt es häufig vor, dass sich der grundlegende Ablauf praktisch nur in Nuancen ändert. Ein Beispiel dafür ist die Übertragung mehrerer Datenbytes innerhalb eines Datenrahmens. In einem naiven Ansatz würde für jedes einzelne zu übertragende Datenbyte ein eigener Protokollabschnitt vorgesehen werden. Diese Abschnitte werden mit den Protokollabschnitten verbunden, die das Verhalten bei Header und Trailer festlegen und beschreiben so die vollständige Übertragung. Ein solches Vorgehen ist mit der hier vorgestellten Methode zwar machbar, bringt jedoch einige Einschränkungen mit sich. Wenn man das Beispiel der Übertragung mehrere Datenbytes weiter verfolgt, wird das besonders deutlich für den Fall, dass die Spezifikation Rahmen mit unterschiedlicher Datenfeldlän-

ge zulässt. Beim naiven Ansatz müsste für jede mögliche Datenfeldlänge ein eigener Zweig von Protokollabschnitten vorhanden sein, in dem das Verhalten für jeweils eine Datenfeldlänge abgebildet wird. So führt dieses Vorgehen im Allgemeinen zu einer unnötig großen Zahl von Protokollabschnitten. Das macht den Satz aller Protokollabschnitte groß und unübersichtlich. Außerdem leidet die Wiederverwendbarkeit der geschriebenen Eigenschaftssätze und die Wahrscheinlichkeit, bei der Überarbeitung der Eigenschaftsbeschreibungen Fehler zu machen, steigt. Es ist deswegen wünschenswert, die Anzahl der Eigenschaftsbeschreibungen möglichst klein zu halten.

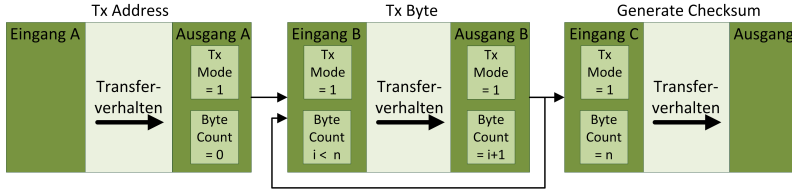


Abbildung 4.10: Wiederholung von Protokollabschnitten

Eine Möglichkeit dafür stellt eine Art von wiederholtem Aufruf eines Protokollabschnittes mit geänderten Parametern dar. Als Beispiel sei hier das oben genannte Senden mehrerer Datenbytes genannt (siehe Abbildung 4.10). Das grundsätzliche Verfahren zur Übertragung bleibt dabei gleich. Die Daten müssen aus dem korrekten internen virtuellen Element übernommen werden und nach einer eventuell notwendigen Verarbeitung auf die externe Datenleitung gesendet werden. Die Unterschiede liegen hier nur in der Position innerhalb des virtuellen Registers, aus dem die Daten entnommen werden. Somit bietet es sich an, die aktuelle Position in einem zusätzlichen virtuellen Register zu speichern. Mit dessen Hilfe können dann die korrekten Daten aus dem virtuellen Datenregister ausgewählt werden. Die Ein- und Ausgangsbedingungen dieses Protokollabschnittes enthalten also die Information zur zu sendenden Anzahl von Datenbytes, die aus dem Header in einem vorangegangenen Protokollabschnitt bestimmt wurden. Zu-

sätzlich wird nun noch das virtuelle Register aufgenommen, in dem die Nummer des zuletzt gesendeten Datenbytes festgehalten wird. Da dieser Protokollabschnitt im weiteren Verlauf zu seinem eigenen Nachfolger werden kann, müssen für diesen Fall die Ein- und Ausgangsbedingungen exakt gleich sein. Am Anfang der Eigenschaftsbeschreibung des Protokollabschnittes muss dann eine Fallunterscheidung getroffen werden, ob ein weiteres Datenbyte zu übertragen ist oder die gewünschte Anzahl erreicht wurde.

Für den ersten Fall wird anhand des virtuellen Registers, in dem die aktuelle Position gespeichert ist das korrekte Datenbyte aus dem virtuellen Datenregister ausgewählt und der korrekte Versand nachgewiesen. Im Anschluss ist nachzuweisen, dass der Wert des virtuellen Positionsregisters korrekt aktualisiert wird. Dieses ist dann die Basis für ein erneutes Durchlaufen des gleichen Protokollabschnittes, um das nächste Datenbyte zu übertragen.

Im zweiten Fall ist die Übertragung mit dem letzten Byte abgeschlossen worden. Dann kann zum nächsten Protokollabschnitt gewechselt werden, in dem zum Beispiel eine Checksumme gebildet oder überprüft wird.

4.5.4 Abbildung des Außenverhaltens eines zu verifizierenden Systems

Neben der Festlegung des in den virtuellen Elementen gespeicherten internen Folgezustandes muss natürlich auch noch das Außenverhalten des Systems festgelegt werden. Dieses Außenverhalten ist durch zwei verschiedene Anteile bestimmt. Zum einen die Signale, die in der Spezifikation explizit vorgegeben werden, weil sie beispielsweise Bestandteil des Busses sind. Bei seriellen Kommunikationssystemen sind das im Allgemeinen die Dateneinbzw. Ausgangsleitungen sowie eine eventuell vorhandene Leitung zur Übertragung eines gemeinsamen Taktsignals.

Zusätzlich sind jedoch auf der Seite des Hostinterfaces noch weitere Signale notwendig, da das betrachtete Businterface im Normalfall nicht völlig autark arbeitet. Im normalen Betrieb ist es üblicherweise so, dass das Businterface

Daten über eine Schnittstelle vom Host entgegen nimmt und dann über den Bus überträgt. Umgekehrt werden Daten vom Bus empfangen und danach dem Host zur Weiterverarbeitung bereit gestellt. Diese Host-Schnittstelle ist, obgleich integraler Bestandteil eines Buscontrollers, häufig in der Spezifikation des Kommunikationsprotokolls nicht näher festgelegt. Um aber dennoch eine Ende-zu-Ende Kommunikation vom Bus zum Host und umgekehrt formal verifizieren zu können, ist es notwendig, eine solche Schnittstelle in der Verifikationsschicht zu definieren. Diese ist dann der eine Endpunkt der Datenübertragung.

Für die Erstellung der Eigenschaftsbeschreibungen ist es erforderlich, dass die Signale und Register einer solchen Host-Schnittstelle bekannt sind. Da jedoch beliebig viele Möglichkeiten existieren, wie eine solche Schnittstelle ausgeprägt werden kann, ist die Wahrscheinlichkeit hoch, dass die in dem allgemeinen Eigenschaftssatz verwendete Darstellung nicht der späteren Realisierung entspricht. So sind hier zum Beispiel sowohl serielle Anbindungen als auch parallele Lösungen denkbar. Die Daten können einzeln in Registern bereit gestellt werden oder in einem adressierbaren Speicher abgelegt werden. Auch die Flußkontrolle kann auf unterschiedlichste Art und Weise realisiert werden.

Um diesem Problem zu begegnen, wurde im Rahmen dieser Arbeit eine einfache parallele Registerschnittstelle festgelegt, die als zweiter Endpunkt für die Daten neben der Busschnittstelle fungiert. Die zu sendenden bzw. empfangenen Daten werden jeweils in einem eigenen Register abgelegt. Ein zusätzliches Valid-Signal kennzeichnet, ob gültige Daten zur Weiterverarbeitung vorhanden sind. Nun könnte man einwenden, dass der auf Basis dieser Schnittstellenbeschreibung erstellte Eigenschaftensatz damit nicht mehr universell auf andere Realisierungen des gleichen Busprotokolls angewendet werden kann. Ähnlich wie schon bei den weiter oben vorgestellten virtuellen Elementen kommt hier allerdings auch wieder die Mappingschicht als verknüpfendes Element zum Einsatz. Mit ihrer Hilfe ist es möglich, eine Anpassung zwischen der für die Erstellung des Eigenschaftensatzes an-

genommenen Schnittstelle und der in der Realisierung gewählten Variante zu vermitteln. Dazu kann es im einfachsten Fall ausreichend sein, die Benennung der virtuellen Elemente an die realen Elemente anzupassen. Komplexere Umsetzungen wären eine seriell/parallel Wandlung oder auch die Abbildung eines komplexeren Protokolls. Näheres dazu wird in Kapitel 4.6 angesprochen. In jedem Fall ist es aber so, dass die Verifikationsschicht für diese Anpassung nicht geändert werden muss. Ein einmal erstellter Satz von Eigenschaftsbeschreibungen, der das Protokoll darstellt, muss also nicht verändert werden, um für eine andere Implementierung Protokollkonformität nachweisen zu können. Dies ist ein wichtiger Beitrag zur Abstraktion der Eigenschaftsbeschreibungen von der Implementierung und der möglichst einfachen Wiederverwendung der Virtualisierungsschicht.

Unter Umständen mag es bei der Verifikation auch sinnvoll sein, nicht das Verhalten des vollständigen Kontrollers zu überprüfen, sondern nur den Teil, der für die Umsetzung des Kommunikationsprotokolls zuständig ist. Damit wäre es auch denkbar, dass den virtuellen Elementen aus der Eigenschaftsbeschreibung interne Signale und Register aus der Implementierung zugewiesen werden.

4.6 Mapping von Protokollgraph und Implementierung

Im vorhergehenden Kapitel wurde die Erstellung der Virtualisierungsschicht beschrieben. Diese liefert einen abstrakten Satz von Eigenschaftsbeschreibungen, der das Protokoll allein aus Sicht der Spezifikation beschreibt. Eine andere, in dieser Arbeit nicht näher betrachtete Aufgabe ist die konkrete Implementierung dieses Protokolls durch einen Entwickler. Sind die Arbeiten auf beiden Seiten weit genug fortgeschritten, können die beiden Schichten zusammen gebracht werden. Während Entwickler und Verifikateur bisher weitestgehend getrennt und unabhängig voneinander gearbeitet haben, stellt die Erstellung der verbindenden Mappingschicht eine gemeinsame Arbeit der beiden dar. Hierzu ist es notwendig, den abstrakten virtuellen Ele-

menten in der Verifikationsschicht reale Elemente aus der Implementierung zuzuordnen. Dies kann zum Beispiel in Form einer Verknüpfungsmatrix dargestellt werden. Mit Hilfe der Mappingschicht ist es dann möglich, den Satz der Eigenschaftsbeschreibungen für das Protokoll und den HDL Code der Implementierung zu einem Design Under Verification (DUV) zusammen zu fassen. Dieses kann dann mit Hilfe der eingesetzten Verifikationswerkzeuge auf Korrektheit hinsichtlich der Spezifikation überprüft werden.

Neben der reinen Zuordnung von realen Elementen zu virtuellen Elementen, die in Kapitel 4.6.1 beschrieben wird, kann es auch notwendig sein, eine Zuordnung von gültigen Wertebereichen für die virtuellen und realen Elemente vorzunehmen. Ein Beispiel dafür ist die unterschiedliche Abbildung von Zeiten in der Spezifikation und in der Implementierung. Überlegungen dazu finden sich in Kapitel 4.6.2.

4.6.1 Mapping der virtuellen Signale/Register/Zustände

Bisher existieren mit der Verifikationsschicht und der Implementierungsschicht zwei voneinander getrennte Darstellungen der Spezifikation. Zur Durchführung der Verifikation müssen der Satz von Eigenschaftsbeschreibungen aus der Verifikationsschicht und der HDL Code der Implementierung zusammengebracht werden. Dazu ist es zwingend notwendig, dass die virtuellen Elemente der Verifikationsschicht mit den realen Elementen der Implementierung verknüpft werden.

Diese Zuordnung wird über die so genannte Mapping-Tabelle abgebildet. Ein Beispiel dafür ist in Abbildung 4.2 zu sehen.

In dieser Tabelle finden sich in der linken Spalte die virtuellen Elemente, die vom Verifikateur für die Erstellung der Eigenschaftsbeschreibungen verwendet worden sind. Diesen werden dann in der nächsten Spalte die entsprechenden realen Elemente zugeordnet.

virtuelles Element	reales Element
Sig_A	*Sig_*1
Sig_B	*Sig_*2(1)
Reg_A(0)	*Reg_*1
Reg_A(1)	*Reg_*2
Reg_B	*Sig_*3 XOR *Reg_*3

Tabelle 4.1: Beispiel für eine Mapping-Tabelle

Während der Verifikateur die von ihm verwendeten virtuellen Elemente kennt und damit direkt in die Tabelle eintragen kann, müssen die realen Elemente aus der Implementierung in einem gemeinsamen Prozess zwischen Verifikateur und Entwickler ergänzt werden. Aufgrund der Konzeption der Methodik ist es so, dass alle in den Eigenschaftsbeschreibungen verwendeten virtuelle Elemente zur Verifikation des Protokollverhaltens zwingend erforderlich sind. Dementsprechend muss für jede Zeile der Mapping-Tabelle eine Zuordnung zu realen Elementen existieren. Auf der anderen Seite hingegen sind nicht alle realen Elemente für die Verifikation erforderlich. Es kann sich dabei zum Beispiel nur um Hilfssignale handeln oder sie werden für die Implementierung von Funktionalitäten benötigt, die nicht Bestandteil der Protokollspezifikation sind. Eine vollständige Auflistung aller in der Implementierung vorhanden realen Elemente macht aus diesem Grund keinen Sinn.

Für die Zuordnung selbst sind verschiedenen Varianten möglich. Diese werden in Abbildung 4.11 verdeutlicht.

Direkte Zuordnung (a) Bei der direkten Zuordnung handelt es sich um den einfachsten Fall. Hier wird einem virtuellen Element nur seine Entsprechung bei den realen Elementen zugewiesen. In diesem Fall handelt es sich also allein um eine Anpassung der Namen.

Abschnittsweise Zuordnung (b) Bei der abschnittsweisen Zuordnung werden dem virtuellen Element nur Teile des zugehörigen realen Element

zugewiesen. Dies ist zum Beispiel dann der Fall, wenn das benötigte virtuelle Signal in der Realisierung Teil eines breiteren Signals ist, in dem auch noch weitere Informationen codiert werden. Dem virtuellen Signal wird dann nur der relevante Teil des realen Signals als Ausschnitt zugewiesen.

Zusammengesetzte Zuweisung (c) Im Gegensatz zur abschnittsweisen Zuordnung kann es auch notwendig sein, einem virtuellen Element mehrere reale Elemente zuzuweisen. Dieser Fall kann dann auftreten, wenn die benötigten Informationen in der Realisierung auf verschiedene Elemente verteilt sind, aber die Informationen doch direkt verwendbar vorliegen. Ein Beispiel dafür wäre ein virtuelles Statusregister in der Virtualisierungsschicht, dessen einzelne Zellen aus verschiedenen Registern der Realisierung gespeist werden.

Transformierende Zuweisung (d) Anders stellt sich der Fall dar, wenn keine direkte Entsprechung zwischen virtuellen und realen Elementen existiert. Dann sind komplexere Transformationen notwendig, um die Entsprechung der Informationen herzustellen. Unter Umständen werden Informationen in der Virtualisierungsschicht und der Implementierungsschicht unterschiedlich codiert (z.B. dezimale Codierung vs. One-Hot-Codierung). Dann ist eine Transformation dieser Informationen durch eine Berechnung notwendig. Denkbar ist auch, dass ein anderer zeitlicher Rahmen für die Darstellung der Informationen gewählt wurde. So könnte sich der Inhalt eines virtuellen Registers als zeitliche Folge eines realen Signals darstellen und umgekehrt. Für diesen Fall sind entsprechende zeitliche Folgen zur Transformation zu verwenden.

Darstellung von virtuellen Zuständen Eine weitere Besonderheit stellen auch die virtuellen Zustände dar. Es ist sehr wahrscheinlich, dass hier keine direkte Zuordnung eines realen Zustandes zu einem virtuellen Zustand möglich ist. Dies ergibt sich schon allein aus der Tatsache, dass keine Vorgaben zur Umsetzung der Spezifikation in Zustandsautoma-

ten gemacht werden. So kann es sein, dass in der Verifikationsschicht nur ein Zustandsautomat verwendet wird, während in der Implementierung mehrere gekoppelte Automaten eingesetzt werden. Analog zu Registern und Signalen kann sich ein Zustand also auch als Kombination mehrerer realer Zustände darstellen.

Die Darstellung eines virtuellen Zustandes ist jedoch nicht nur auf reale Zustände beschränkt. So kann sich der Zustand eines System auch über Werte von realen Signalen oder Registern zu festgelegten Zeitpunkten bestimmen lassen. Deswegen kann es notwendig sein, auch diese Größen in die Darstellung der virtuellen Zustände aufzunehmen.

4.6.2 Definition gültiger Werte und Wertebereiche

In einem ersten Schritt wurde in Kapitel 4.6.1 beschrieben, wie den virtuellen Elementen die realen Elemente zugeordnet werden können. Neben dieser rein namensbasierten Zuordnung ist es jedoch auch erforderlich, gültige Werte und Wertebereiche für die Elemente festzulegen. Erst damit lässt sich das Verhalten eindeutig festlegen. Ähnlich wie bei der Erstellung der Mapping-Tabelle ist auch hier eine intensive Zusammenarbeit zwischen dem Entwickler und dem Verifikateur erforderlich, da diese ihr jeweiliges Detailwissen beisteuern müssen.

Um eine freie Zuordnung zwischen den Werten der virtuellen und den realen Elementen zu erlauben, ist es wichtig, dass bei der Erstellung der Eigenschaftsbeschreibungen für die Verifikationsschicht keine konkreten Zahlen und Werte verwendet werden. Stattdessen sollten benannte Platzhalter in der Art von Variablen oder Konstanten zur Anwendung kommen. Diesen Platzhaltern können nun in der Mappingschicht konkrete Werte aus der Implementierung zugeordnet werden. Auch diese Zuordnung kann wieder in Form einer Tabelle dargestellt werden (Beispiel siehe Abbildung 4.2). Jedem

in der Verifikationsschicht eingesetzten virtuellen Element sollte damit mindestens ein Platzhalter mit realen Werten zugeordnet werden.

virtuelles Element	**Zuweisung**
bit_duration	16
valid_bit_duration	[14..18]
sleep_timeout	[128256]
state_idle	$main\_fsm = idle; receive\_fsm = wait\_for\_bit$

Tabelle 4.2: Beispiel für eine Tabelle zur Wertezuordnung

In der ersten Spalte befinden sich die in der Verifikationsschicht verwendeten Platzhalter. In den weiteren Spalten werden für jedes damit verknüpfte reale Elemente (diese Informationen können aus der Mapping-Matrix übernommen werden), die gültigen Wert oder Wertebereich angegeben. Damit sollte also jedem Platzhalter ein Satz konkreter Werte zugeordnet werden.

Ein Grund für die Notwendigkeit dieses Vorgehens liegt darin, dass Werte aus der Spezifikation für die Darstellung in Eigenschaftsbeschreibungen und in der Implementierung unterschiedlich umgesetzt werden können. Ein Beispiel dafür ist die Beschreibung von Zeitpunkten oder Zeiträumen. Diese werden in der Spezifikation üblicherweise in Sekunden oder Millisekunden angegeben. Bei einer Implementierung hingegen wird man sich im Allgemeinen auf ein Signal bekannter Periode, wie zum Beispiel das Taktsignal als Referenz zur Bestimmung von Zeiten beziehen. Die Dauer einer Millisekunde wird dann zum Beispiel in Vielfachen der Taktperiode angeben. Zeiten lassen sich damit über einen Zähler messen, der vom Takt angesteuert wird. Somit wird einer Zeitdauer ein bestimmter Zählerstand zugewiesen. An dieser Stelle ist man damit allerdings sehr implementierungsspezifisch und nicht mehr unabhängig. So können sich die verwendeten Taktraten zwischen verschiedenen Implementierungen genauso unterscheiden wie die Umsetzung in den Zähler und dessen Abfrage. Hier kommt wieder die Mappingschicht zum Tragen, die die Abbildung zwischen den abstrakten Werten aus der Spezifikation zu realen Werten vornimmt.

Neben der Zuweisung eines festen realen Wertes für einen Platzhalter ist es auch möglich, dort erlaubte Wertebereiche zu hinterlegen. Diese werden dann in die Eigenschaftsbeschreibungen als zulässige Wertebereiche übernommen. Dies ist zum Beispiel für die weiter oben betrachtete Abbildung des Zeitverhaltens notwendig. So sind in der Spezifikation im Allgemeinen keine exakten Zeitpunkte für das geforderte Eintreten eines Ereignisses angegeben, sondern Bereiche, in denen eine bestimmte Aktion erwartet wird. Diese zeitliche Varianz drückt sich dann in einem Bereich aus, in dem sich der für die Bestimmung der Zeit verwendete Zähler befinden darf.

Außerdem können Wertebereiche aus der Implementierung selbst herrühren, obwohl sie in dieser Form in der Spezifikation selbst nicht vorgesehen sind. So könnte die Spezifikation einen Timeout festlegen, bis zu dem eine Aktion abgeschlossen sein muss. In der Implementierung wird dafür ein Zähler eingesetzt, der so lange hochgezählt wird, bis die Aktion abgeschlossen ist. Die Abfrage in der Verifikationsschicht, ob der Timeout noch nicht erreicht wurde, wird damit in einen erlaubten Wertebereich des Zählers umgesetzt.

Eine weitere Besonderheit stellt die Darstellung von Zuständen der Verifikationsschicht mit den realen Elementen dar. Oftmals ist es so, dass sich die in der Verifikationsschicht angewendeten Zustandsautomaten deutlich von denen der Implementierung unterscheiden. Dies lässt sich nicht vermeiden, da ja gerade eine möglichst große Unabhängigkeit zwischen der Arbeit der Verifikateurs und des Entwicklers gewünscht wird. Damit kommt es auch nicht zu Absprachen, die in einer ähnlichen Darstellung münden. Besonders augenscheinlich ist das für den Fall, dass die Verifikationsschicht schon existiert und auf eine andere Implementierung angewendet werden soll. Die Verifikationsschicht ist in diesem Fall schon vollständig und wird nicht mehr verändert werden, kann also auch nicht der Implementierung nachgebaut werden.

Um nun trotzdem die beiden Schichten miteinander verbinden zu können, müssen diese Unterschiede in der Mappingschicht aufgefangen werden. Für

den Fall der Zustände kann das bedeuten, dass einem virtuellen Zustand nicht nur ein einziger konkreter Zustand der Implementierung zugewiesen wird, sondern eine Menge von Zuständen aus mehreren in der Implementierung vorhandenen FSMs. Unter Umständen kann es auch notwendig sein, nicht nur Zustände, sondern auch Signale und Registerwerte in die Definition eines virtuellen Zustandes mit aufzunehmen, wenn diese zur Bestimmung des exakten Systemstatus notwendig sind.

4.7 Umsetzung und iterative Durchführung der Verifikation

In den vorangegangenen Schritten wurde die Implementierung der verschiedenen Schichten des Modells beschrieben. Mit dem Satz von Eigenschaftsbeschreibungen in der Verifikationsschicht, der HDL Beschreibung der Realisierung in der Implementierungsschicht und der verbindenden Elemente in der Mappingschicht sind nun alle Komponenten zur formalen Verifikation der Implementierung zumindest in ersten Versionen vorhanden.

In einem nächsten Schritt können diese nun an ein Tool zu formalen Verifikation weitergegeben werden. Dieses wendet die erstellten Eigenschaftensätze auf den HDL Code an und überprüft, ob die geforderten Eigenschaften eingehalten werden. Im Idealfall treten dabei keine Gegenbeispiele auf und die Implementierung kann als konform mit der Eigenschaftsbeschreibung angesehen werden.

Im Allgemeinen wird es jedoch so sein, dass bei der Durchführung der Eigenschaftsprüfung Gegenbeispiele gefunden werden. Deshalb setzt hier ein iterativer Prozess ein, in dem die bisher erstellten Schichten schrittweise korrigiert und verfeinert werden. Während die Analyse möglicher Gegenbeispiele wieder in gemeinsamer Arbeit von Entwickler und Verifikateur durchgeführt werden, könnte die gegebenenfalls notwendige Korrektur auch wieder unabhängig von jedem der beiden alleine bewerkstelligt werden. In der Praxis ist der Abschnitt der iterativen Verfeinerung jedoch ein gemeinsamer Prozess.

Gegenbeispiele können generell drei mögliche Ursachen haben. Sie können auf Fehler in der Implementierung, bei der Erstellung der Eigenschaftsbeschreibungen oder auch auf eine fehlerhafte Spezifikation selbst zurückzuführen sein. Die genaue Fehlerursache gilt es nun in gemeinsamer Arbeit von Entwickler und Verifikateur herauszuarbeiten. Die Basis bildet dabei die Diskussion über das Verständnis der Spezifikation bei den beiden beteiligten Partnern. Da Spezifikationen in vielen Fällen noch in natürlicher Sprache verfasst sind, lassen sie dementsprechend auch Spielraum für Interpretationen. Wird also eine Abweichung von dem durch die Eigenschaftsbeschreibung erwarteten Verhalten und dem Verhalten der Implementierung durch ein Gegenbeispiel aufgezeigt, so ist zu entscheiden, wie die entsprechende Passage der Spezifikation korrekt zu interpretieren ist. Dementsprechend ist dann eine Anpassung der Eigenschaftsbeschreibung oder der Implementierung notwendig, um einen fehlerfreien Durchlauf ohne Gegenbeispiel zu erreichen. Dieser Prozess ist nicht in eine Methodik zu fassen, hier ist die Erfahrung der Beteiligten Personen gefragt. Spätestens hier bietet es sich auch an, den Kreis der beteiligten Personen zu erweitern, um weitere Experten mit ihrer Expertise hinzu zu ziehen.

Besonders in den frühen Stadien der Entwicklung der Eigenschaftsbeschreibungen kann eine Konsequenz aus dieser Analyse sein, das die Eigenschaftsbeschreibungen überarbeitet werden müssen. Oft bedeutet das eine Verfeinerung einzelner Protokollabschnitte, um zum Beispiel mehr Details zum gewünschten Verhalten aufzunehmen. So können zum Beispiel die Ein- und Ausgangsbedingungen ergänzt werden, um zusätzliche relevante virtuelle Elemente in die Beschreibung des Protokollabschnittes aufzunehmen. Dabei ist jedoch immer darauf zu achten, dass diese so allgemein wie möglich bleiben und keine Spezifika aus der konkreten Implementierung aufgenommen werden.

Eine deutlich größere Quelle von Gegenbeispielen stellt jedoch die Mappingschicht dar, da mit ihrer Hilfe die Anpassung der Implementierung an die Verifikationsschicht vorgenommen wird. Im Idealfall eines schon häufiger

für die Verifikation herangezogenen Eigenschaftensatzes ist dieser als konstant und ausgereift anzusehen, da er die Spezifikation fehlerfrei wiedergibt. Gegenbeispiele werden in diesem Fall häufig dadurch erzeugt, dass die realen Elemente nicht sauber auf die virtuellen Elemente abgebildet werden. Durch die fehlende Berücksichtigung von Randbedingungen der Implementierung kommt es zu *False-Negatives*, da Gegenbeispiele generiert werden, die im fehlerfreien Betrieb beim Start aus dem Reset-Zustand nicht erreicht werden können. Diese zusätzlichen Randbedingungen müssen dann in die Mappingschicht mit aufgenommen werden. Gerade bei mehreren nebenläufigen FSMs kann es sein, dass die Auswirkungen der einzelnen FSM auf den aktuellen Zustand nicht berücksichtigt werden.

Zusätzliche Quellen von Gegenbeispielen bieten auch die oft in einer Implementierung enthaltenen Zusatzfunktionen. Neben den durch die Spezifikation des Kommunikationsprotokolls vorgegebenen Funktionalitäten sind häufig auch weitere Funktionen implementiert, die nur am Rande mit der eigentlichen Datenübertragung zu tun haben. Dies könnte zum Beispiel eine Konfigurationsfunktion sein, mit der die Implementierung bei Start oder auch im laufenden Betrieb umkonfiguriert werden kann. Erfolgt die Umkonfiguration zu einem verbotenen Zeitpunkt während einer Übertragung, so wird das Tool zur Eigenschaftsprüfung ein Gegenbeispiel liefern. Auch für diesen Fall ist die Mappingschicht zu ergänzen, um diese aus einer anderen Spezifikation herrührenden Einschränkungen wiederzugeben.

Im Idealfall kommt eine ausgereifter Satz von Eigenschaftsbeschreibungen zum Einsatz, der schon mit unterschiedlichen Implementierungen fehlerfrei angewendet wurde. In diesem Fall ist davon auszugehen, dass Gegenbeispiele entweder aus einer fehlerhaften Implementierung oder aus einer nicht ausreichenden Adaption über die Mappingschicht herrühren. Der Satz der Eigenschaftsbeschreibungen stellt dann eine feste Referenz dar, gegen die die Implementierung verifiziert wird. Werden alle Eigenschaftsbeschreibungen fehlerfrei durchlaufen, so ist davon auszugehen, dass die Implementierung ebenso der Spezifikation entspricht.

4.8 Berücksichtigung funktionaler Aspekte

Schon im Kapitel 4.5.3 wurde darauf hingewiesen, dass es bestimmte Verhaltensweisen bei Kommunikationsprotokollen gibt, die nicht innerhalb eines einzigen Protokollabschnittes behandelt werden können. Dies sind unter anderem Übertragungen, bei denen auf eine Anfrage hin eine Antwort des Teilnehmers erfolgen soll. Beispiele für solche Übertragungen sind die Anforderung von Status und Diagnoseinformationen sowie zur Konfiguration von Busteilnehmern über den Bus selbst. Die Komplexität bei diesen Übertragungen liegt darin, dass die Zeit, die zwischen einer Anfrage und deren Beantwortung liegt in der Spezifikation nicht immer klar vorgegeben ist. Unter Umständen können nach einer Anfrage sogar zuerst anderen Übertragungen stattfinden und danach erst die Antwort auf die ursprüngliche Anfrage gesendet werden. Dieses Verhalten der Unterbrechung und späteren Wiederaufnahme von Übertragungen ist auch als *Split Transactions* bekannt. Zwischen dem Beginn einer solchen Transaktion und Ihrer Beendigung liegen also theoretisch beliebig viele weitere Transaktionen. Damit ist diese Transaktion praktisch nicht mehr innerhalb einer einzigen Eigenschaftsbeschreibung zu erfassen, da diese ja auch jedes mögliche Verhalten zwischen Beginn und Ende abbilden müsste. In diesem Kapitel soll ein Ansatz vorgestellt werden, mit dem solche *Split Transactions* dennoch in Form von Eigenschaftsbeschreibungen erfasst werden können.

Eine der Grundideen beim Erstellen von Eigenschaftsbeschreibungen für Protokollabschnitte aus dem Protokollgraphen ist die Tatsache, dass jeweils die Eingangs- und Ausgangsbedingungen klar und eindeutig definiert sein müssen. Das ist notwendig, um die einzelnen Protokollabschnitte sauber miteinander zu einer Kette verknüpfen zu können. Damit dies funktioniert, muss also das erwartete Verhalten innerhalb eines Protokollabschnittes eindeutig festgelegt sein. Dann ist eine Überprüfung mit Hilfe einer einzigen Eigenschaftsbeschreibung möglich. Anders stellt sich das jedoch im Fall der *Split Transactions* dar. Hier ist das Verhalten des Systems zwischen der Eingangs-

und der Ausgangsbedingung nicht eindeutig festgelegt. Im einfachsten Fall handelt es sich nur um eine beliebig lange Wartezeit, in der das System ruht und auf die Wiederaufnahme der Übertragung wartet. Im anderen Fall kann es aber auch sein, dass nach Übertragung der Anfrage beliebige weitere Übertragungen stattfinden. Deren Art und Anzahl ist im Vorfeld nicht bekannt und kann deswegen nicht in der einen Eigenschaftsbeschreibung abgebildet werden.

Um solche *Split Transactions* dennoch im Protokollgraphen erfassen zu können, ist eine Aufteilung der Transaktion auf mehrere Eigenschaftsbeschreibungen notwendig. Die Idee dahinter ist es, die Verifikation der *Split Transaction* in drei Phasen aufzuteilen. In der ersten Phase wird überprüft, ob die Anfrage korrekt verarbeitet wurde. Zu dieser Verarbeitung gehören neben dem korrekten Protokollverhalten auf dem Datenbus auch die Speicherung der Anfrage mit allen benötigten Informationen, um daraus später eine korrekte Antwort generieren zu können. Zu diesem Zweck wird ein zusätzliches virtuelles Register definiert, dass die relevanten Informationen bereit hält. Dies beinhaltet zum einen natürlich die Information selbst, dass eine Transaktion auf Beantwortung wartet. Zusätzlich könnten aber auch weitere Informationen wie die Art der Transaktion und die angefragten Daten hinterlegt werden.

Die Informationen in diesem virtuellen Register dürfen im normalen Betrieb nur bei der Behandlung einer *Split Transaktion*, also beim Empfang der Anfrage oder bei dem Senden der Antwort verändert werden. Die Abarbeitung von gewöhnlichen Datenübertragungen sollte keine Auswirkungen auf dieses virtuelle Register haben. Deswegen muss in der nächsten Phase sicher gestellt werden, dass der Inhalt des virtuellen Registers in allen anderen Protokollabschnitten unverändert bleibt. Dazu ist in jedem anderen Protokollabschnitt zu zeigen, dass der Wert des virtuellen Registers, dass für die Speicherung der Anfrage zuständig ist, innerhalb des Protokollabschnittes nicht verändert wird. Weitere Annahmen zu den Werten des virtuellen Statusregisters sind im Allgemeinen nicht erforderlich. Ausnahmen von dieser

Unveränderbarkeit sind allerdings möglich. So ist es naheliegend, dass ein „Reset“ auch ein Rücksetzen des virtuellen Statusregisters erzwingt. Weiterhin sind natürlich auch Konstellationen denkbar, in denen die Spezifikation Gründe für den Abbruch oder die Veränderung einer eingeleiteten *Split Transaktion* vorgibt. Dies muss gegebenenfalls in weiteren Protokollabschnitten berücksichtigt werden.

In der letzten Phase wird schließlich die korrekte Ausführung der Antwort auf die ursprüngliche Anfrage der *Split Transaction* nachgewiesen. Dies geschieht wieder in einem eigenen Protokollabschnitt. Innerhalb der dem Protokollabschnitt zugeordneten Eigenschaftsbeschreibung ist dann zu zeigen, dass auf Basis der in dem virtuellen Statusregister gespeicherten Informationen eine korrekte Antwort auf den Bus gelegt wird. Dabei muss das virtuelle Register jedoch nicht alleinige Quelle für die Informationen sein, die übermittelt werden. So können natürlich auch aktuell vorhandene Daten übertragen werden, die nicht im virtuellen Statusregister gespeichert waren, wenn die Spezifikation das vorsieht.

Die hier vorgestellte Aufteilung in drei Phasen ermöglicht es, *Split Transactions* mit beliebigem Verhalten zwischen einer Anfrage und deren Beantwortung abzubilden und formal verifizierbar zu machen. Insbesondere unbekannte Wartezeiten zwischen Anfrage und Antwort lassen sich so effizient abbilden. Die Komplexität der verwendeten Eigenschaftsbeschreibung kann klein gehalten werden, da sie jeweils nur ein kleines Zeitfenster abbilden müssen. Die beliebig lange Zeitdauer ergibt sich dann durch die Aneinanderkettung der einzelnen Protokollabschnitte, die einzeln verifiziert wurden. Die Komplexität bleibt dabei konstant, da ein einmal verifizierter Protokollabschnitt nicht erneut verifiziert werden muss.

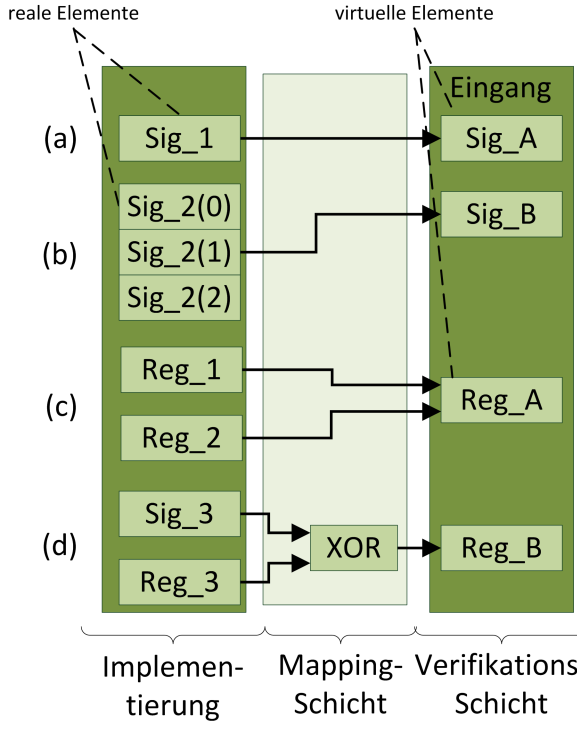


Abbildung 4.11: Verschiedene Möglichkeiten des Mappings von realen zu virtuellen Elementen

5 Fallbeispiel LIN

5.1 Kommerzieller Intellectual Property Core (IP-Core) für den LIN-Bus

Als Anwendungsbeispiel für die in dieser Arbeit beschriebene Methodik ist ein industrieller Local Interconnect Network (LIN) Core zum Einsatz gekommen. Es handelt sich hier um ein kommerzielles Produkt, das weltweit in großer Anzahl erfolgreich eingesetzt wird. Der Core wurde firmenintern intensiv getestet und besteht den LIN-Conformance-Check. Bei diesem Test und auch im praktischen Einsatz konnte kein Fehler in der Implementierung festgestellt werden.

Bei dem LIN Controller handelt es sich um ein autonomes Modul, dass zur Kommunikation über den LIN-Bus verwendet werden kann, ohne dass dazu ein externer Mikrocontroller benötigt wird. Der Controller unterstützt den LIN Data Link Layer in den Versionen 1.3, 2.0 oder 2.1. In dieser Arbeit wurde allerdings nur die Version 2.0 des LIN-Protokolls betrachtet. Implementiert wurde der Controller in VHDL auf RTL Ebene. Der Quellcode stand für die Verifikation zur Verfügung.

Der Controller wird über vier miteinander gekoppelte Zustandsautomaten (Finite State Machines (FSMs)) gesteuert, die jeweils für Teilaspekte des Betriebes verantwortlich sind. Er kann damit völlig autonom und ohne einen zusätzlichen externen Mikrocontroller betrieben werden. Auf Grund der nötigen Überabtastung des Bussignals ist die interne Taktfrequenz des Controllers deutlich größer als die Baudrate des LIN Busses. Eine Anpassung dieser unterschiedlichen Taktraten ist über konfigurierbare Prescaler möglich.

Dadurch kann die Taktfrequenz beliebige Werte über der systembedingten Mindestfrequenz annehmen. Für das Versenden und den Empfang von Daten stehen interne Nachrichtenspeicher (message buffer) zur Verfügung. Abhängig von der gewählten Version des LIN Protokolls bietet der Controller auch Unterstützung für benutzerspezifische Übertragungen und den Diagnosemodus des LIN-Busses. Außerdem ist eine Steuerung des Sleep-Mode und der dadurch nötigen Wake-Up Prozedur integriert.

Der IP-Core ist in weiten Grenzen sowohl während der Entwurfszeit als auch im späteren Betrieb konfigurierbar. Die Konfiguration kann dabei auf drei verschiedene Arten erfolgen. Zur Entwurfszeit können grundlegende Eigenschaften wie die zu unterstützende Version des LIN Protokolls und die Anzahl der Nachrichtenspeicher festgelegt werden. Diese sind nach Fertigung des Cores nicht mehr veränderbar. Eine weitere Möglichkeit der Konfiguration stellen im Controller vorhandene Konfigurationseingänge dar. Über diese kann auch später das Verhalten des Controllers geändert werden. So ist es unter anderem möglich, den Protected Identifier (PID) zu ändern, unter dem der Controller angesprochen werden kann. Auch lassen sich darüber einzelne Message Buffer ein- und ausschalten und die Länge der zu übertragenden Botschaften verändern. Diese Änderungen dürfen jedoch nicht während des laufenden Betriebes, sondern nur vor Systemstart vorgenommen werden. Zusätzlich ist es möglich, einzelne Parameter im laufenden Betrieb über Diagnose-Botschaften des LIN Busses zu ändern.

Zum Datenaustausch kommen auf der Host-Seite die schon weiter oben angesprochenen Message Buffer zum Einsatz. Über diese werden die empfangenen und zu sendenden Daten in paralleler Form bereit gestellt. Die Gültigkeit der Daten wird über zusätzliche Handshake-Leitungen gesichert.

Zur Durchführung der Verifikation des LIN IP-Cores wurden aus praktischen Gründen einige Annahmen und Einschränkungen getroffen, die jedoch die Aussagekraft nicht wesentlich einschränken. So wurde eine feste Konfiguration des Controllers auf das LIN 2.0 Protokoll mit einer konstanten Anzahl von Message Buffern eingestellt. Auch alle beim Start konfigurierba-

ren Parameter wurden auf einen festen Wert gestellt, um die Anzahl der zu betrachtenden Varianten klein zu halten. Die grundlegende Funktionalität bleibt davon unberührt.

Eine wesentliche Einschränkung stellt die Annahme von korrektem Verhalten der Umgebung an den Schnittstellen des zu verifizierenden Controllers dar. Dazu wird angenommen, dass sich alle Teilnehmer auf dem Bus jederzeit korrekt verhalten und aktiv keine ungültigen Signale auf den Bus senden. Gleiches gilt auch für die Host-Seite. Ein fehlerhaftes Signal auf dem Bus kann daher nur vom Controller selbst herrühren. Außerdem wird stets von idealen Signalen ohne Störungen wie zum Beispiel Glitches oder Timingverschiebungen ausgegangen.

Einer der Gründe für diese Entscheidung ist, dass das Verhalten im Fehlerfall nicht in der LIN Spezifikation vorgegeben ist. Eine Aussage zur korrekten Implementierung der Fehlerbehandlung ist also nicht möglich, da diese vom Entwickler frei gewählt werden kann. Eine Überprüfung des Fehlerverhaltens würde also auf die Nachbildung des im Core realisierten Verhaltens hinauslaufen, was nicht im Sinne einer von der Implementierung unabhängigen Verifikation sein kann. Weiterhin geht es in dieser Arbeit darum, korrektes Protokollverhalten nachzuweisen. Das Verhalten des Cores bei Verletzung der in der Spezifikation definierten Parameter gehört in den Bereich der Robustheitsuntersuchung und ist nicht Teil dieser Arbeit. Sehr wohl untersucht werden jedoch die in der Spezifikation beschriebenen Fehlerfälle wie zum Beispiel den Empfang einer fehlerhaften PID oder eines falschen Prüfsumme.

5.2 Protokollzerlegung des LINs

In diesem Kapitel soll die Anwendung der in Kapitel 4 dargestellten Methodik auf ein konkretes Fallbeispiel gezeigt werden. Dazu sollte die korrekte Implementierung des LIN Protokolls in der Version 2.0 [1] durch den im vorangegangenen Abschnitt beschriebenen IP-Core nachgewiesen werden.

Da der IP-Core schon vollständig implementiert war, beschränkt sich diese Arbeit auf die Erstellung der Verifikations- und der Mappingschicht. Der erste Schritt zur Erstellung der Verifikationsschicht ist die Zerlegung des Protokolls in einzeln handhabbare Protokollabschnitte. Diese können danach durch individuelle Eigenschaftsbeschreibungen abgebildet werden.

In der Abbildung 5.1 ist dazu noch einmal die grundlegende Struktur eines LIN Datenrahmens wiedergegeben. Er besteht aus einem Request fester Länge, der immer von der Master Task im System gesendet wird und einer Response variabler Länge, die von der adressierten Slave Task verschickt wird. Mit Hilfe des Request bestimmt der Master Task, welcher anderen Knoten an der Kommunikation beteiligt ist und welche Reaktion von ihm erwartet wird. Der Beginn einer Übertragung ist durch das Break-Signal gekennzeichnet, auf dessen Besonderheiten im Kapitel 5.5 näher eingegangen wird. Außerdem dient der Request der Synchronisierung der Takgeneratoren zwischen den beiden Kommunikationspartnern. Die Oszillatoren können im Laufe des Betriebs abweichen. Vor Beginn der Datenübertragung müssen diese Abweichungen ermitteln und gegebenenfalls ausgeglichen werden. Danach folgt der Protected Identifier (PID), mit dem Ziel und Art des Transfers festgelegt werden. In der Response schließlich sendet die angesprochene Slave Task die Informationen, die sich aus dem empfangenen PID ergeben. Abgeschlossen wird die Übertragung mit einer Checksumme und einer Ruhepause auf dem Bus.

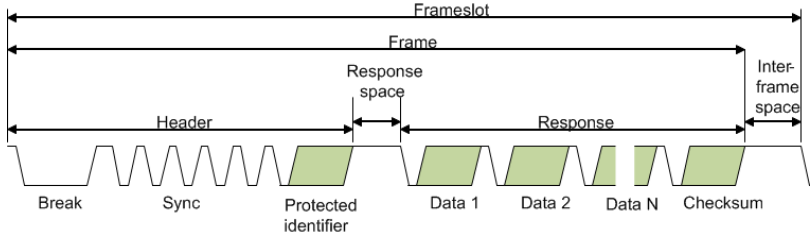


Abbildung 5.1: Struktur eines LIN Datenrahmens

Es bieten sich nun verschiedene Möglichkeiten zur Festlegung der Grenzen zwischen einzelnen Protokollabschnitten an. Die Darstellung des kompletten Rahmens in einer einzigen Eigenschaftsbeschreibung widerspricht dem Grundgedanken der vorgestellten Methodik. Sie soll unter anderem durch die Anwendung von *Divide et Impera* die Komplexität serieller Protokolle handhabbar machen. Eine Zerlegung des Protokolls in einzelne Bits als anderes Extrem erscheint auch wenig sinnvoll. Eine individuelle Beschreibung jedes Bits mit den jeweiligen Eingangs- und Ausgangsbedingungen bedeutet einen unverhältnismäßig großen Overhead. Außerdem würde es auf diese Art und Weise sehr komplex, Fallunterscheidungen vorzunehmen, da sich diese im Protokoll im Allgemeinen nicht auf einzelnen Bits, sondern auf längere Folgen wie zum Beispiel den PID bezieht. Um dies abzubilden wäre, es erforderlich, die einzelnen Protokollabschnitte durch die Verwendung zusätzlicher virtueller Register miteinander zu koppeln.

Eine andere Möglichkeit der Aufteilung stellt die Trennung in Abschnitte für den Request und die Response dar. Diese würden dann jeweils als eine Einheit in einer Eigenschaftsbeschreibung abgebildet werden. Damit reduziert sich die Anzahl de Protokollabschnitte deutlich im Vergleich zum bitweisen Vorgehen. Außerdem lassen sich einfach Bit-übergreifende Eigenschaften beschreiben. Nachteilig an dieser Granularität der Aufteilung ist jedoch, dass dadurch eine relativ hohe Redundanz beider Beschreibung der einzelnen Protokollabschnitte besteht. So muss zum Beispiel für jeden Protokollabschnitt, der eine Request-Variante beschreibt das immer wieder gleiche Verhalten zur Erkennung eines Breaks und der Synchronisation betrachtet werden. Auch für jede denkbare Response muss ein eigener Protokollabschnitt definiert werden.

Bei genauerer Betrachtung des Protokolls stellt sich heraus, dass sich der Rahmen mit Ausnahme des Break Feldes aus einzelnen Feldern der Länge eines Bytes zusammen setzt. Diese Länge bietet sich damit als „natürliche“ Länge für die einzelnen Protokollabschnitte an. Die Beschreibung des Protokolls orientiert sich weitgehend an diesen Byte-Grenzen, so dass die

einzelnen Protokollabschnitte möglichst unabhängig voneinander behandelt werden können. Gleichzeitig stellen die Aufteilung in Bytes einen praktikablen Kompromiss zwischen der Anzahl und der Komplexität der Protokollabschnitte dar.

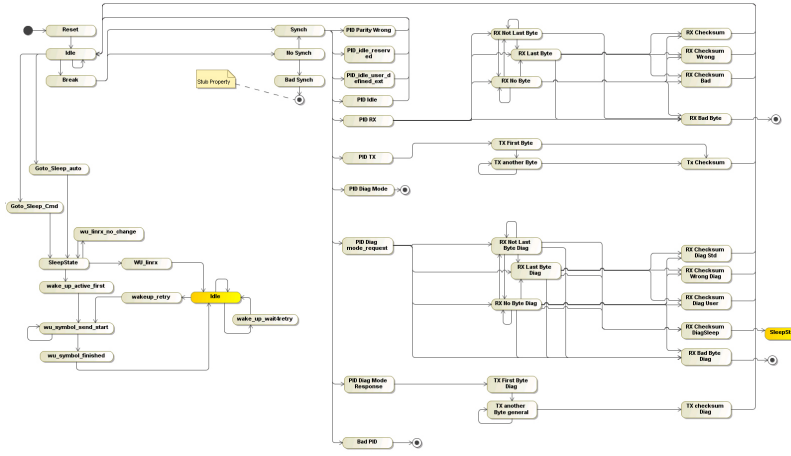


Abbildung 5.2: Protokollgraph des LIN-Protokolls (LIN 2.0)

Der Protokollgraph, der sich aus diesen Überlegungen ergibt, ist in Abbildung 5.2 dargestellt. Ausgehend von einem Reset mit dem zugehörigen Protokollabschnitt erreicht der Graph den Ruhezustand „Idle“. Eine Übertragung wird durch das Senden des Break-Signals eingeleitet, gefolgt von dem Byte zur Synchronisation. Danach folgt die Übertragung des PID. Wie man aus der Abbildung entnehmen kann, ist dies auch der zentrale Verzweigungspunkt für das Protokoll, da sich je nach empfangenem PID das Verhalten der Kommunikationsteilnehmer unterscheidet. Grundsätzlich kann dabei zwischen dem Senden und dem Empfangen von Botschaften unterschieden werden. Weiterhin lassen sich die Botschaften in gewöhnliche Datenpakete und solche zur Diagnose und Konfiguration unterteilen. Das Senden und Empfangen von normalen Datenpaketen wird über die Ketten PID-

TX und PID-RX abgebildet. Für die Pakete des Diagnosemodus kommen die Pfade PID-Diagmode_request und PID-Diagmode_response zum Einsatz. Die weiteren Pfade behandeln im wesentlichen Fehlerfälle. Während die meisten Pfade wieder zurück zum „Idle“ Zustand führen, gibt es andere Pfade, die in der graphischen Darstellung in einem eingekreisten Punkt enden und nicht mehr zurück führen. Es handelt sich dabei um die schon in Kapitel 4 erwähnten Fälle, bei denen ein Fehlerverhalten aufgetreten ist, für das in der Spezifikation keine korrekte Vorgehensweise vorgegeben ist. Für das Auftreten eines solchen Falls kann also keine Eigenschaftsbeschreibung mit dem gewünschten Verhalten erstellt werden. Um alle möglichen Fälle abzudecken, müssen diese Protokollabschnitte jedoch zumindest als Rumpf in den Protokollgraphen aufgenommen werden.

Eine weitere, von der Datenübertragung weitgehend unabhängige Kette, stellt die Überprüfung des so genannten „Sleep mode“ dar. Durch ihn ist es möglich, den Controller in einen stromsparenden Schlafmodus zu schicken. Dies geschieht nach dem Empfang eines speziellen Datenpaketes über den Bus. Alternativ wird der „Sleep mode“ auch dann aktiviert, wenn es für eine vordefinierte Zeitspanne keine Aktivität auf dem Bus mehr gegeben hat. Neben dem Nachweis, dass der Sleep Mode korrekt erreicht wird, muss auch nachgewiesen werden, dass er bei Busaktivität innerhalb einer vorgegebenen Zeit wieder verlassen wird.

In den folgenden Abschnitten soll nun die Anwendung der in Kapitel 4 beschriebenen Methodik anhand von exemplarischen Beispielen dargestellt werden.

5.3 Entkopplung

Einer der zentralen Punkte in der in dieser Arbeit beschriebenen Methodik ist die Entkopplung der Verifikation von der Implementierung. Die Eigenschaftsbeschreibungen, die das gewünschte Verhalten des Protokolls wiedergeben, sollen dabei möglichst unabhängig von der tatsächlichen Imple-

mentierung sein. Eine Zusammenführung der beiden Ebenen geschieht erst durch die Einführung der Mapping-Schicht. In dieser Schicht werden den virtuellen Elementen die realen Elemente der Implementierung zugewiesen.

Ein Beispiel, an dem sich dies sehr gut zeigen lässt, ist die Entkopplung der Zustandsdarstellung bei der Verifikation und der Implementierung. In der Verifikationsschicht wird das Protokollverhalten in Form einer einzigen FSM dargestellt. Sie entspricht dem Protokollgraphen aus Abbildung 5.2. Nebenläufigkeiten existieren in dieser Darstellung nicht. Das komplette Verhalten des Protokolls wird als Abfolge einzelner Schritte dargestellt.

Im Falle einer Implementierung des Protokolls innerhalb eines IP-Cores muss das jedoch nicht zwangsläufig auch der Fall sein. So kann es zum Beispiel möglich sein, dass noch weitere FSM existieren, die weitere Funktionalitäten innerhalb der IP realisieren. Diese können, müssen aber nicht unbedingt direkt mit der eigentlichen Protokoll FSM verknüpft sein. Genauso ist es natürlich denkbar, dass Protokollverhalten selbst nicht mit Hilfe einer einzigen FSM, sondern durch mehrere gekoppelte FSM darzustellen. Die Funktionalität, die damit realisiert wird, kann damit auch deutlich über das in der Spezifikation vorgegebene Maß hinausgehen.

Im Beispiel der hier betrachteten kommerziellen IP war die Funktionalität mit Hilfe von insgesamt vier gekoppelten FSM realisiert. Jede FSM ist dabei für einzelne Aspekte des Verhaltens zuständig. Das Protokollverhalten der IP ergibt sich aus dem Zusammenspiel dieser unterschiedlichen FSM, so dass es nicht möglich war, eine FSM herauszunehmen und einzeln gegen die Protokoll-FSM zu verifizieren. Damit ergab sich also die schon im Methodik-Kapitel angesprochene Notwendigkeit, die einzelnen Zustände der Protokoll-FSM mit dem unterschiedlichen Zuständen der einzelnen realen FSM zu verknüpfen. Für jeden virtuellen Zustand der Protokoll-FSM war deswegen eine Zuordnung zu den realen Zuständen erforderlich.

5.4 Split Transactions

Beim Betrieb eines Kommunikationssystems können zwei unterschiedliche Varianten von Datenübertragungen unterschieden werden. Bei einem singulären Transfer wird eine Datenübertragung gestartet und ohne Unterbrechung durch andere Transfers beendet. Dies entspricht dem Fall einer normalen Datenübertragung beim LIN-Bus, bei der die Master-Task einen Transfer initiiert, der gleich drauf von einer Slave Task ausgeführt wird. Danach ist diese Übertragung abgeschlossen und es kann ein neuer, davon unabhängiger, Transfer eingeleitet werden. Ein solcher Transfer ist mit der in dieser Arbeit beschriebenen Methodik relativ einfach abzubilden, da es sich hier um eine bekannte Sequenz einzelner Eigenschaftsbeschreibungen handelt. Das gewünschte Verhalten des zu verifizierenden Modell ist im Vorfeld bekannt und kann mit den bisher beschriebenen Methoden modelliert werden.

Einen zweiten Fall stellen jedoch die in dieser Arbeit so genannten Split Transactions dar. Bei diesen Transfers ist es so, dass zwischen dem Initiieren einer Übertragung und deren Abschluss beliebig viel Zeit liegen kann. Zusätzlich ist es auch möglich, dass in dieser Zeit weitere Übertragungen stattfinden, die vom ersten Transfer unabhängig sind. Dies ist im konkreten Anwendungsbeispiel des LIN-Busses bei den Diagnosebotschaften der Fall. Hier übermittelt die Mastertask in einem ersten Schritt eine Anfrage an eine andere Slave Task. Diese muss dann zu einem späteren Zeitpunkt mit der zur ursprünglichen Anfrage passenden Rückmeldung antworten. Mit dieser Funktion kann zum Beispiel die Konfiguration und der aktuelle Zustand eines Busteilnehmers abgefragt werden. Da die Spezifikation keine Vorgaben zum Zeitpunkt der Antwort macht, ist es theoretisch möglich, dass nach der Anfrage noch beliebige andere Transfers stattfinden, bevor die Antwort zur ursprünglichen Anfrage abgerufen wird.

Unter diesen Voraussetzungen ist das gewünschte Verhalten des Systems nicht mehr eindeutig festgelegt. Zwischen Beginn und Ende einer solchen Split Transaction kann also nur eine kurze Wartezeit liegen oder aber belie-

big viele weitere Übertragungen. Um alle Möglichkeiten abdecken zu können, müsste also für jeden Fall eine eigene Kette von Eigenschaftsbeschreibungen erstellt werden, die den jeweiligen Fall beschreibt. Zwar ließe sich durch den Aufbau der Kette in Form eines Baums die Komplexität möglicherweise noch etwas reduzieren. Es bleibt aber trotzdem unmöglich, alle Kombinationen abzubilden.

Wie schon in Kapitel 4.8 beschrieben, wurde in dieser Arbeit deswegen eine Methode entwickelt, um solche Fälle trotzdem mit der vorgestellten Methodik handhaben zu können. Dazu wird innerhalb einer Eigenschaftsbeschreibung nachgewiesen, das die Anfrage korrekt empfangen und als solche im System gespeichert wurde. Dazu kam im Fall der Abbildung des Diagnose-Verhaltens ein virtuelles Register zum Einsatz. In diesem Register wird die Botschaft gespeichert, die bei der zweiten Abfrage der Master-Task gesendet werden soll. Bei dem virtuellen Register handelt es sich in diesem Fall um ein Macro (`generate_diag_tx_fullvector`), dass in Abhängigkeit vom Typ der Anfrage aus den anderen im System vorhandenen virtuellen Registern und Zuständen die vollständige Antwortbotschaft zusammenstellt. Dies dient zum einen der Abstraktion der Beschreibung der geforderten Eigenschaften von der tatsächlichen Implementierung. Zum anderen erhöht es aber auch deutlich die Lesbarkeit der erstellten Eigenschaftsbeschreibungen und hilft so zu einem einfacheren Verständnis derselben. Zusätzlich kann es so möglich sein, Fehler in der Beschreibung zu vermeiden oder zumindest einfacher zu entdecken.

In einer ersten Eigenschaftsbeschreibung wird also nachgewiesen, dass das zu testende Modell die Anfrage korrekt empfängt und, dass die für die korrekte Antwort nötigen virtuellen Elemente richtig gesetzt werden. Diese Eigenschaftsbeschreibung ist damit im Allgemeinen die einzige Stelle im gesamten Protokollgraphen, an der die dazu verwendeten virtuellen Elemente geändert werden dürfen. Eine mögliche Ausnahme ist die Eigenschaftsbeschreibung zum Senden der Antwort. Hier wäre es denkbar, dass die virtuellen Elemente auf einen anderen Wert zurück gesetzt werden.

In allen anderen Eigenschaftsbeschreibungen muss nun sichergestellt werden, dass die in der ersten Eigenschaftsbeschreibung gesetzten virtuellen Elemente unverändert bleiben. Dazu wird zu Beginn des betrachteten Zeitfensters jeder anderen Eigenschaftsbeschreibung der Wert dieser virtuellen Elemente bestimmt. Für das Ende des betrachteten Zeitfensters wird dann bewiesen, dass die Werte am Ende denen am Anfang entsprechen. Da davon ausgegangen wird, dass die einzelnen Eigenschaftsbeschreibungen atomar sind und zur gleichen Zeit keine weiteren, durch andere Eigenschaftsbeschreibungen erfassten Vorgänge ablaufen, spielt es auch keine Rolle, ob die betrachteten virtuellen Elemente innerhalb des betrachteten Zeitfensters andere Werte annehmen. Wichtig ist nur, dass die ursprünglichen Werte am Ende wieder hergestellt werden. Durch diesen Zweiten Schritt ist nun also gewährleistet, dass alle für das Erstellen der Antwort der Split Transaction notwendigen virtuelle Elemente immer unverändert bleiben. Damit können nun also nach der Verarbeitung der Anfrage beliebige weitere Eigenschaftsbeschreibungen verknüpft werden, ohne das Ergebnis der Split Transaction zu verändert.

Den Abschluss der Behandlung von Split Transactions stellt für den hier beschriebenen Anwendungsfall eine weitere Eigenschaftsbeschreibung dar, die das Versenden der korrekten Antwort auf die Anfrage der Mastertaks nachweist. In dieser Eigenschaftsbeschreibung wird das vorher beschriebene virtuelle Register verwendet, um die tatsächlich gesendeten Daten mit den geforderten Werten zu vergleichen. Auch hier erweist sich die von der Realisierung unabhängige Darstellung als virtuelles Register als hilfreich, da die Darstellung innerhalb des virtuellen Registers exakt den Datenbytes entspricht, wie sie auf dem Bus erwartet werden. Die abschließende Eigenschaftsbeschreibung lässt sich damit sehr eingängig beschreiben, da in ihr nur die auf dem Bus erscheinenden Daten mit dem Inhalt des virtuellen Registers verglichen werden müssen.

Mit Hilfe des hier beschriebenen Vorgehens konnte für den in dieser Arbeit als Anwendungsbeispiel betrachteten Core das korrekte Diagnosever-

halten in allen Kombinationen nachgewiesen werden, was mit bisher verwendeten Ansätzen, mit das vollständige Verhalten beschreibenden Eigenschaftsbeschreibungen, nicht möglich war. Der zusätzliche Aufwand für die Umsetzung der Methodik beschränkt sich dabei auf die Erstellung der Eigenschaftsbeschreibungen zur Überprüfung des korrekten Setzen und Abfragens der virtuellen Elemente. Zusätzlich mussten in alle weiteren Eigenschaftsbeschreibungen zwei Blöcke eingefügt werden, mit denen die Werte der verwendeten virtuellen Elemente ausgelesen und nachgewiesen wurde, dass sie am Ende der einzelnen Eigenschaftsbeschreibungen nicht verändert wurden. Diese Blöcke werden sinnvollerweise ebenfalls durch Macros gebildet, deren Inhalt während des iterativen Erstellens des Eigenschaftsgraphens immer wieder verfeinert werden. Somit können die bisher erstellten Eigenschaftsbeschreibungen unverändert bleiben, während immer mehr Funktionalität ergänzt wird.

Neben der ursprünglichen Ausrichtung auf die Behandlung von Split Transactions stellte sich im Laufe der Arbeiten heraus, dass die hier vorgestellte Methodik auch zur Verifikation einer weiteren Besonderheit des LIN-Busses verwendet werden kann und zwar für die Beschreibung des Sleep Mode. Das ursprüngliche Einsatzgebiet, für das der LIN-Bus entwickelt wurde, ist der Automobilbereich. Eine besonders wichtige Eigenschaft stellt in diesem Umfeld der Energieverbrauch der eingesetzten Komponenten dar. Dieser beeinflusst zum einen während des Betriebes den Gesamtverbrauch des Automobils, wirkt sich aber vor allem auf die Zeit aus, die ein Auto abgestellt bleiben kann, ohne dass der Akku über den Motor nachgeladen werden muss. Ein niedrigerer Ruhestromverbrauch verlängert diese Zeit. Aus diesem Grund sollten alle im Automobil eingesetzten Steuergeräte und auch die verwendete Kommunikationsinfrastruktur auf einen möglichst geringen Ruhestromverbrauch hin optimiert sein. Ein probates Mittel dafür ist die Realisierung eines Ruhezustandes (engl. Sleep Mode), in dem keine Daten verarbeitet werden und der Energieverbrauch verringert wird, indem zum Beispiel nicht benötigte Komponenten abgeschaltet werden.

Auch der LIN-Bus unterstützt einen solchen Ruhezustand, den Sleep Mode. Im Sleepmode werden keine Botschaften mehr versendet und die beteiligten Geräte reduzieren ihren Energiebedarf so weit wie möglich. Für den Buscontroller bedeutet das, dass im Idealfall nur noch die Teile aktiv bleiben, die für das Verlassen des Ruhezustandes erforderlich sind.

Der Ruhezustand kann durch zwei verschiedene Auslöser aktiviert werden. Zum einen existiert in der Spezifikation ein spezielles Kommando, dass alle an den Bus angeschlossenen Knoten in den Ruhezustand schickt. Zum anderen überwacht jeder Knoten des Buses selbstständig auf Aktivitäten. Nach Ablauf einer in der Spezifikation vorgegebenen Zeitspanne (mind. 4s, höchstens 10s) von Inaktivität auf dem Bus muss die Slavetask selbstständig in den Sleep Mode wechseln. Verlassen wird der Sleep Mode durch den Empfang eines so genannten `Wake-up` Signals, bei dem der Bus für mindestens 150µs auf den dominaten Pegel wechselt. Nach weiteren 100ms muss der Knoten wieder auf Anfragen auf dem Bus reagieren können.

Die Besonderheit bei der Verifikation dieses Verhaltens liegt in den vergleichsweise großen Zeitspannen, die es zu betrachten gilt. Mit dem bisher angewendeten Verfahren, bei dem das Verhalten in einer einzigen Eigenschaftsbeschreibung abgebildet wird, ergeben sich auf Grund der Größe des zu betrachtenden Zeitfensters sehr lange Laufzeiten. Diese machen die Verifikation nicht unmöglich, sie verzögern sie aber.

Auch für diese Problemstellung hat sich die in dieser Arbeit vorgestellte Methodik als hilfreich erwiesen. Ähnlich wie bei einer Split Transaction wird die Verifikation des Sleep Mode auch in drei Phasen unterteilt. Zuerst wird in einer Eigenschaftsbeschreibung nachgewiesen, dass die Notwendigkeit des Ruhezustandes korrekt erkannt und dieser dann auch eingenommen wird. Die beinhaltet auch das Setzen aller virtuellen Elemente, die später für das korrekte Aufwachen erforderlich sind.

Das Auslösen über das `Goto-Sleep` Kommando lässt sich wie eine normale Datenübertragung behandeln. Es wird ein Paket mit dem entsprechenden Kommando empfangen, woraufhin der Core in den Sleep Mode wechselt.

Anders sieht es jedoch bei der Auslösung des Sleep Mode durch das Erreichen des Time-Outs bei Inaktivität aus. Hier kommt ein virtuelles Register als Zähler zum Einsatz, dass die Zeit der Inaktivität auf dem Bus zählt. Befindet sich der Core im Ruhezustand, so muss der Zähler erhöht werden. Dies wird darüber nachgewiesen, dass in der Eigenschaftsbeschreibung, die dem Idle-Zustand zugeordnet ist, der virtuelle Zähler inkrementiert wird. Umgekehrt muss in allen anderen Eigenschaftsbeschreibungen, die den normalen Betrieb abbilden, nachgewiesen werden, dass der Wert des virtuellen Zählers wieder zurück gesetzt wird. Der Zählerstand, bei dem der Sleep Mode aktiviert werden soll, ist nun wiederum Annahme bei der Eigenschaftsbeschreibung, die das Sleep-Verhalten beschreibt. Damit ergibt sich wieder eine Kette aus einer Verhaltensbeschreibung für eine Übertragung, gefolgt von einer Iteration der Beschreibung des Idle-Verhaltens, solange der virtuelle Zähler unter dem Maximalwert ist. Wird der Maximalwert erreicht, greift die nächste Eigenschaftsbeschreibung zur Verifikation des Sleep-Verhaltens.

Mit der hier vorgestellten Methodik muss damit die Eigenschaftsbeschreibung des Idle-Modes nur ein einziges Mal bewiesen werden, um beliebig lange Wartezeiten abbilden zu können. Die Laufzeit des gesamten Beweises ist damit konstant und nicht mehr wie bisher abhängig von der Größe des betrachteten Zeitfensters.

5.5 Mehrwert der formalen Verifikation am Beispiel des Breakverhaltens

Während der Arbeiten mit dem kommerziellen Core wurde auch ein Fehler entdeckt, der durch alle bisher durchgeführten Tests nicht entdeckt wurde. Es handelt sich dabei um einen Fehler bei der Erkennung des Break Signals, der unter ungünstigen Timing-Bedingungen auftreten kann. Um den Fehler nachvollziehen zu können, soll dazu zuerst die Problematik der Break-Erkennung erläutert werden, um dann den Fehlerfall und dessen Erkennung darzustellen.

Wie kurz in Kapitel 2.6.1 angesprochen, wird jede Datenübertragung auf dem LIN-Bus von der Master Task durch ein so genanntes Break-Field eingeleitet. Es handelt sich dabei um einen dominanten Pegel, der mindestens 13 nominale Bitzeiten lang konstant gehalten wird. Abgeschlossen wird das Break-Field durch einen rezessiven Pegel der Länge einer nominalen Bitzeit. Das Break-Field kennzeichnet den Beginn einer neuen Übertragung und beendet damit unmittelbar alle eventuell noch laufenden Transfers. Nach der Spezifikation muss jede Slave Task das Break-Field jederzeit nach spätestens elf Bitzeiten erkennen. Eine Slave Task, die das Break-Field nicht korrekt erkennt, verpasst damit auch den dadurch gestarteten Transfer.

Ein stark vereinfachtes Beispiel der Eigenschaftsbeschreibung zur Verifikation des Break-Verhaltens ist in Listing 5.1 dargestellt. Zu Beginn wird deklariert, dass kein Reset erfolgen soll und die Konfiguration stabil ist. Ebenso sollen alle Register gültige Werte enthalten. Es handelt sich also um den normalen Betrieb des Cores. Im Assume-Teil wird dann angenommen, dass ein korrektes Break-Field auf dem Bus anliegt. Beim 16-fachen Oversampling des untersuchten Cores entspricht das 208 Zeitschritten mit dominantem Pegel. Danach wechselt das Bussignal wieder auf den rezessiven Pegel. Im Beweisteil soll dann gezeigt werden, dass das Break-Field korrekt erkannt wurde und der Core sich deswegen in dem Zustand befindet, in dem er auf das folgende Sync-Field wartet.

```
1  PROPERTY break IS
2  DEPENDENCIES:
3    no_reset,
4    configuration_stable;
5
6  ASSERTIONS:
7    all_states_valid,
8    all_registers_valid;
9
10 FOR TIMEPOINTS:
11   t_break_start = t+4,
12   t_rx_up = t_break_start+208;
13
14 ASSUME:
```

```
15    DURING[t_break_start-4, t_break_start-1]: lin_rx = '1';
16    DURING[t_break_start, t_rx_up-1]: lin_rx = '0';
17    DURING[t_rx_up, t_rx_up+4]: lin_rx = '1';
18
19  PROVE:
20    AT t_rx_up+4:
21      fsm_eval_syncfield,
22      registers_eval_syncfield;
23
24    DURING[t_rx_up, t_rx_up+4]: lin_tx = '1';
25
26  END PROPERTY;
```

Listing 5.1: Break-Eigenschaftsbeschreibung

Beim Überprüfen der Eigenschaftsbeschreibung wurde nun jedoch ein Gegenbeispiel generiert, das in Abbildung 5.3 zu sehen ist. Wie man aus der Abbildung entnehmen kann, wurde das Break-Field zwar im Zyklus 810 erkannt, der Core befindet sich jedoch weiterhin im Ruhezustand `prtfsm_idle`. Der Grund für dieses Gegenbeispiel soll im Folgenden dargestellt werden.

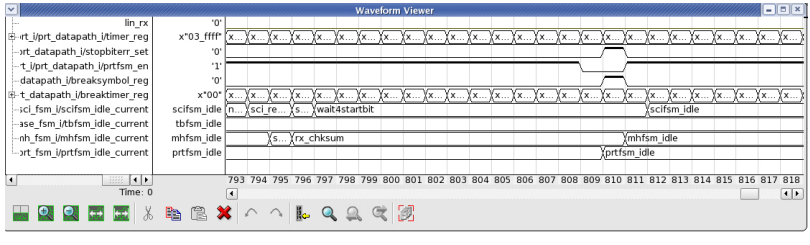


Abbildung 5.3: Gegenbeispiel

Die LIN-Spezifikation [1] erlaubt eine Abweichung von bis zu ±15% der aktuellen Taktfrequenz im Vergleich zur nominalen Taktfrequenz. Das die Frequenz der Übertragung auch die Position der Abtastzeitpunkte beeinflusst, muss die aktuell verwendete Taktfrequenz zu Beginn einer Übertragung ermittelt werden. Zu diesem Zweck sendet die Master Task nach dem Break-Field das so genannte Sync-Field, das aus einer alternierenden Folge von

Nullen und Einsen besteht (0x55). Dieses Feld kann von den Slave Tasks verwendet werden, um die aktuelle Übertragungsfrequenz zu bestimmen und sich daran anzupassen. Zu diesem Zweck hat der in dieser Arbeit betrachtete Core ein internes Register (`timebase_reg`), in dem der zur Korrektur nötige Wert gespeichert wird. Allerdings kann dieser Wert erst nach dem Empfang des Sync-Fields bestimmt und angewendet werden. Für die Erkennung des Break-Fields wird deswegen in der vorliegenden Implementierung ein fester Standardwert für die Abtastzeitpunkte verwendet.

Zur Erklärung des Gegenbeispiels wird folgendes Szenario angenommen. Es findet eine Übertragung mit einem so genannten „slow sync“ statt. Das bedeutet, dass die Dauer eines Bits länger ist als die nominale Dauer. Dies ist in Abbildung 5.4 dargestellt. Entsprechend der Annahme in der Eigenschaftsbeschreibung wird nun ein konstanter dominanter Pegel auf den Bus gelegt, der dem Senden eines Break-Fields durch die Master Task entspricht.

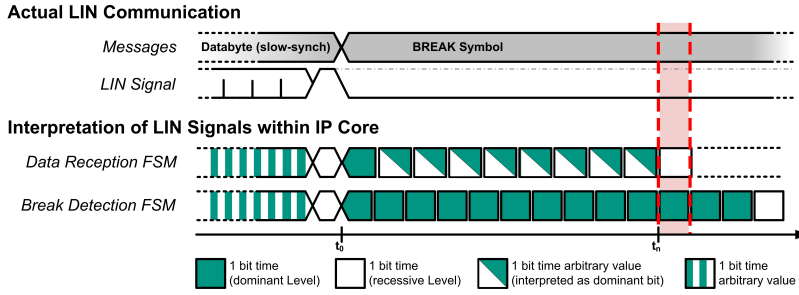


Abbildung 5.4: Fehlerhafte Break-Erkennung

Im Core befinden sich mehrere parallele FSM, von denen eine die auf dem Bus anliegenden Daten einliest. Zur Unterscheidung der einzelnen Bits kommt dabei das `timebase_reg` Register zum Einsatz. Gleichzeitig ist eine weitere FSM aktiv, die das Bussignal nach dem Break-Field absucht. Diese verwendet zur Abtastung die nominale Bitzeit und nicht den durch das `timebase_reg`

korrigierten Wert. Damit interpretieren diese beiden FSM eine unterschiedlichen Anzahl von Bits in das empfangene Bussignal.

Die für den Empfang von Daten zuständige FSM interpretiert das Signal auf dem Bus als Folge aus einem Startbit und acht dominanten Datenbits. Als zehntes Bit würde sie nun ein rezessives Stopbit erwarten. Dies ist jedoch nicht der Fall, da der Bus weiterhin auf dominantem Pegel bleibt. Dementsprechend erkennt die FSM einen Stopbit-Fehler und signalisiert diesen mit dem internen Flag `stopbiterr_set`, das allerdings nur für einen Taktzyklus gesetzt bleibt.

Gleichzeitig interpretiert die für die Erkennung des Break-Signals zuständige FSM, aufgrund der unterschiedlichen Zeitbasen, den immer noch dominanten Pegel als elftes Bit eines Break-Fields. Dies wird wiederum durch das Setzen des internen Flags `breaksymbol_reg` angezeigt. Auch diese Flag ist nur für einen einzigen Taktzyklus aktiv. Zu sehen ist das in Abbildung 5.3 im Zyklus 810.

Diese Flags werden im nächsten Taktzyklus ausgewertet. Auf Grund eines Fehlers in der Implementierung des Cores erhält dabei das Flag zur Erkennung des Stopbit-Fehlers eine höhere Priorität als die Erkennung des Break-Fields. Obwohl also das Break-Field ursprünglich korrekt erkannt wurde, bricht der Core in Folge des Stopbit-Fehlers die aktuelle Übertragung ab und geht zurück in den Ruhezustand. Dort wartet er auf das Auftreten eines Break-Fields, mit dem eine neue Übertragung gestartet werden soll. Die Information, dass schon ein Break-Field vorlag, ist verloren gegangen, da das entsprechende Flag nach einem Taktzyklus wieder zurück gesetzt wurde.

Der Core hat also das aktuelle Break-Field nicht wie gefordert erkannt und kann nicht auf die anstehende Datenübertragung reagieren. Diese Daten sind verloren und müssen erneut übertragen werden. Das Verhalten des Cores entspricht damit nicht der Spezifikation.

Nachdem der Fehler erkannt und die Ursache ermittelt war, wurde er zur Korrektur an die Entwicklungsabteilung weitergeleitet. Der Fehler wurde

behoben und der korrigierte Quellcode wieder mit der unveränderten Eigenschaftsbeschreibung auf Fehler untersucht. Auch in diesem zweiten Durchlauf war es jedoch so, dass die Eigenschaft nicht bewiesen werden konnte. Statt dessen wurde ein neues Gegenbeispiel erzeugt. Bei einer erneuten Analyse des Gegenbeispiels stellte sich heraus, dass es sich hierbei um einen neuen Fehler handelte, der in die gleiche Klasse einzuordnen war, wie der vorher entdeckte Fehler. Auch hier trat wieder zeitgleich mit der Erkennung des Break-Fields ein anderer Protokollfehler auf. Da auch dieser Fehler gegenüber der Erkennung des Break-Fields priorisiert wurde, erfolgte die Behandlung des Break-Fields nicht korrekt. Damit war auch in diesem Fall der aktuell gesendete Datenrahmen verloren.

Die Entdeckung dieser beiden Fehler in diesem intensiv getesteten und vielfach eingesetzten Core demonstriert die Leistungsfähigkeit der formalen Verifikation. Mit simulativen Ansätzen ist es erforderlich, Annahmen zu möglichen Fehlerszenarien zu machen. Danach entwirft man einen Testcase um nachzuweisen, dass genau dieser Fehlerfall nicht eintrifft. Um einen anderen Fehlerfall abzudecken, ist wiederum ein neuer Testcase notwendig. Im konkreten Fall wären also mindestens zwei verschiedene Testcases notwendig gewesen, um diese Fehler entdecken zu können. Vor allem aber ist es erforderlich, dass der Verifikateur eine Vorstellung hat, welche Fehlerszenarien möglich sind. Im Gegensatz dazu ermöglicht es die formale Verifikation, sich von diesen Fehlerszenarien zu lösen. Statt dessen wird das gewünschte Sollverhalten vorgegeben und dessen Einhaltung durch die aktuelle Implementierung nachgewiesen. Somit war es im vorliegenden Anwendungsbeispiel möglich, mit Hilfe einer einzelnen Eigenschaftsbeschreibung unterschiedliche und von einander unabhängige Fehlerfälle entdecken zu können.

Es waren keinerlei Annahmen zu möglichen Fehlerszenarien erforderlich, da nur das Sollverhalten entsprechend der Spezifikation in Eigenschaftsbeschreibungen umgesetzte werden musste.

6 Zusammenfassung und Ausblick

In modernen vernetzten Systemen, wie sie unter anderem auch im automobilen Umfeld verwendet werden, spielt die Kommunikation zwischen den einzelnen Teilen des Systems eine immer bedeutendere Rolle. Große Teile der Funktionalität sind nur durch die Kommunikation mehrerer Steuergeräte und Sensoren möglich. Aufgrund der heute üblichen Verteilung der Entwicklung der einzelnen Komponenten auf unterschiedliche Zulieferer es ist besonders wichtig, eine einheitliche Kommunikationsschnittstelle sicher zu stellen. Hier ist der Einsatz von Techniken der formalen Verifikation sehr vielversprechend, da er beweisbare Aussagen zum Verhalten eines Systems ermöglicht.

Die Eigenschaftsbeschreibungen zur Festlegung des gewünschten Verhaltens eines zu untersuchenden Systems werden dabei aktuell so formuliert, dass sie spezifisch für eine konkrete Implementierung sind. Wird die Implementierung geändert, so müssen auch die Eigenschaftsbeschreibungen geändert werden. Neben dem Mehraufwand besteht hierbei auch die Gefahr, dass die beiden Sätze von Eigenschaftsbeschreibungen nicht das gleiche Verhalten beschreiben.

Als Lösung für dieses Problem wird in der vorliegenden Arbeit eine Methodik vorgestellt, die es ermöglicht, die Erstellung der Eigenschaftsbeschreibungen unabhängig von einer konkreten Implementierung zu machen. Dazu werden in den Eigenschaftsbeschreibungen so genannte virtuelle Register, Zustände und Ein- und Ausgänge verwendet, die sich alleine aus der Interpretation der Spezifikation ergeben. Damit gibt der Satz von Eigenschaftsbeschreibungen alleine das gewünschte Verhalten vor, wie es in der

Spezifikation definiert wurde. Diese in der vorliegenden Arbeit so genannte Verifikationsschicht ist damit unabhängig von einer tatsächlichen Implementierung.

Um nun dennoch die Verbindung zu einer konkreten Implementierung (diese wird als Implementierungsschicht bezeichnet) herzustellen, kommt die so genannte Mappingschicht zum Einsatz. In ihr werden den abstrakten virtuellen Registern, Zuständen sowie Ein- und Ausgängen die realen Pendants aus der Implementierung zugeordnet. Dadurch wird der Bezug zwischen den Eigenschaftsbeschreibungen und der Implementierung hergestellt.

Diese Unabhängigkeit von der Implementierung bietet mehrere Vorteile. Werden die Eigenschaftsbeschreibungen mit einer konkreten Implementierung im Hinterkopf erstellt, besteht immer die Gefahr, dass diese so erstellt werden, dass sie zur Implementierung passen. Unter Umständen werden damit aber Fehler aus der Implementierung in die Eigenschaftsbeschreibungen übernommen. Bei dem hier vorgestellten Ansatz erfolgen die Erstellung der Eigenschaftsbeschreibungen und die Implementierung im Idealfall unabhängig von einander durch zwei verschiedene Personen oder Teams. Diese interpretieren jeweils die Spezifikation und setzen sie nach ihrem Verständnis um. Sollten hier einzelne Teile unterschiedlich interpretiert werden, so sollte das beim Zusammenfügen der beiden Teile und dem Abarbeiten der Eigenschaftsbeschreibungen durch Gegenbeispiele aufgedeckt werden. Für diese Differenzen muss dann abgeklärt werden, wie die Spezifikation tatsächlich zu verstehen ist. Auf diese Art und Weise können also Fehler, die sich aus unterschiedlicher Interpretation der Spezifikation ergeben, vermieden werden.

Neben der Möglichkeit, Interpretationsfehler aufzudecken, bietet der vorgestellte Ansatz auch die Möglichkeit einer Wiederverwertung der erstellten Eigenschaftsbeschreibungen. Die Eigenschaftsbeschreibungen sind aufgrund ihrer Konstruktion implementierungsunabhängig. Erst durch die Mappingschicht wird die Abbildung auf eine konkrete Implementierung vorgenommen. Damit wird es möglich, die Eigenschaftsbeschreibungen als Kon-

formitätstest für ein Protokoll zu verwenden. Verschiedene Implementierungen, die die Eigenschaftsbeschreibungen erfüllen, zeigen damit auch das gleiche Außenverhalten in Bezug auf das Protokoll. Wenn das Protokoll selbst vollständig und fehlerfrei beschrieben ist, kann damit eine fehlerfreie Zusammenarbeit der verifizierten Implementierungen garantiert werden.

Neben der Abstraktion ist ein weitere Beitrag dieser Arbeit die Erweiterung bekannter Verifikationsmethodiken nach dem Teile-und-herrsche-Ansatz um die Besonderheiten serieller Protokolle. Bei serieller Kommunikation treten lange zu betrachtende Zeitfolgen auf, die zu großen Laufzeiten der verwendeten Verifikationstools führen. Diese großen Laufzeiten sind beim Abarbeiten des vorhandenen Satzes von Eigenschaftsbeschreibungen eher unkritisch. Diese Test können ohne Benutzerinteraktion automatisiert ablaufen. Jeder Test wird dabei nur einmal durchgeführt, so dass die Gesamtlaufzeit beschränkt ist.

Anders sieht es hingegen beim Erstellen der Eigenschaftsbeschreibungen aus. Dieser Prozess ist im Allgemeinen ein interaktiver, iterativer Prozess, bei dem der Verifikateur Schritt für Schritt die Eigenschaftsbeschreibungen verbessert und verfeinert. Nach jedem Iterationsschritt muss die Eigenschaftsbeschreibung wieder vom Verifikationswerkzeug abgearbeitet werden. Verzögerungen hier wirken sich direkt auf die Zeit aus, die zum Erstellen der finalen Eigenschaftsbeschreibungen benötigt wird.

Zu diesem Zweck wird das Protokoll in kleine Einheiten zerlegt, die jeweils nur einen kleinen Teil der Funktionalität abbilden. Diese Einheiten werden dann mit einander verknüpft, um den kompletten Umfang des Protokollverhaltens darzustellen. Da jede Einheit klein gehalten wird, ergeben sich kurze Laufzeiten und somit schnelleres Arbeiten beim iterativen Erstellen der Eigenschaftsbeschreibungen.

Ein weitere Vorteil des hier vorgestellten Teile-und-herrsche-Ansatzes ist es, dass damit auch einfach Vorgänge erfasst werden können, bei denen die Dauer zwischen auslösender Bedingung und Ergebnis nicht im Voraus bekannt ist. Dazu wird in dieser Arbeit ein dreiphasiger Ansatz dargestellt, bei

dem zuerst von den auslösenden Bedingungen her nachgewiesen wird, dass alle Vorkehrungen getroffen wurden, um die korrekte Reaktion formulieren zu können. In der zweiten Phase wird dann nachgewiesen, dass alle an der Erzeugung der korrekten Antwort beteiligten Informationen in weiteren Schritten nicht verändert werden, um dann in der dritten Phase die Erzeugung der korrekten Antwort auf die ursprüngliche Anfrage nachzuweisen.

Die in dieser Arbeit vorgestellte Methodik wurde anhand eines kommerziellen LIN-IP-Cores evaluiert. Es wäre nun wünschenswert, weitere IP-Cores und auch verschiedene Bussysteme zu betrachten, um den Nutzen der Methodik besser bewerten zu können. Hieraus können sich auch Erfahrungen ergeben, wie die abstrakten Eigenschaftsbeschreibungen der Verifikationsschicht zu beschreiben sind, damit die Anpassung durch die Mappingschicht möglichst einfach vorgenommen werden kann.

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

Literatur- und Quellennachweis

[1] *LIN Specification Package Revision 2.0*.

[2] *LIN Specification Package Revision 2.2*.

[3] BIERE, CIMATTI und C. ET AL: *Bounded Model Checking*. Advances in Computers Volume 58, Academic Press, 2003.

[4] BORMANN, J.: *Vollständige funktionale Verifikation*. Doktorarbeit, Universität Kaiserslautern, 2009.

[5] BORMANN, J., S. BEYER, A. MAGGIORE, M. SIEGEL, S. SKALBERG, T. BLACKMORE und F. BRUNO: *Complete formal verification of Tricore2 and other processors*. In: *Design and Verification Conference (DVCon)*, 2007.

[6] BRYANT, R. E.: *Graph-Based Algorithm for Boolean Function Manipulation*. In: *IEEE Transactions on Computers*, S. 677–691, August 1986.

[7] DUDEN: *Duden Definition zum Begriff Ëchtzeit"*. `http://www.duden.de/rechtschreibung/Echtzeit`. Zuletzt aufgerufen 04.2014.

[8] GMBH, O. S.: *OneSpin 369 User Documentation*, August 2008.

[9] GORAI, S., S. BISWAS, L. BHATIA, P. TIWARI und R. S. MITRA: *Directed-simulation assisted formal verification of serial protocol and bridge*. Design Automation Conference, 2006 43rd ACM/IEEE, S. 731–736, 0-0 2006.

[10] GRZEMBA, A. und H.-C. VON DER WENSE: *LIN-Bus*. Franzis, 2005.

[11] IABG: *V-Modell*. `http://v-modell.iabg.de/index.php`. Zuletzt aufgerufen 04.2014.

[12] KIMMESKAMP, T., M. JOCHIM, J. FORMANN, K. ECHTLE, S. BULACH

und K. WEINBERGER: *Formale Verifikation eines komplexen seriellen Kommunikationsprotokolls - „Lessons Learned“ am Beispiel einer FlexFay-IP-Verifikation*. In: VDE/VDI/GMM (Hrsg.): *Zuverlässigkeit und Entwurf*, S. 159. VDE Verlag GmbH, 2008.

[13] KÖLBL, R. H.: *Falsches Testen - Teure Fehler*. `http://www.elektronikpraxis.vogel.de/themen/elektronikmanagement/projektqualitaetsmanagement/articles/118963/`. Zuletzt aufgerufen 04.2014.

[14] LIN-KONSORTIUM: *LIN - the cost-competitive serial communication system for localized vehicle electrical networks*. `http://www.lin-subbus.org/`. Zuletzt aufgerufen 07.2009.

[15] MOLINA, A. und O. CADENAS: *Functional Verification: Approaches and Challenges*. Latin American Applied Research, 37:65–69, 2007.

[16] MOORE, G. E.: *Cramming more components onto integrated circuits*. Electronics, 38(8):114 ff., April 19 1965.

[17] NGUYEN, D.-M.: *System-on-Chip Protocol Compliance Verification Using Interval Property Checking*. Doktorarbeit, Technische Universität Kaiserslautern, 2009.

[18] ONESPIN SOLUTIONS GMBH: *OneSpin-Webseite*. `http://www.onespin.de`. Zuletzt aufgerufen 07.2009.

[19] ROYCE, W. W.: *Managing the Development of Large Software Systems*. In: *IEEE WESCON*, August 1970.

[20] WIKIPEDIA: *Wikipedia-Website zum OSI-Modell*. `http://de.wikipedia.org/wiki/OSI-Modell?`. Zuletzt aufgerufen 10.2013.

[21] WIKIPEDIA: *Wikipedia-Website zur Lieferantenpyramide*. `http://commons.wikimedia.org/wiki/File:Lieferantenpyramide.jpg`. Zuletzt aufgerufen 07.2014.

[22] YANG, Y.-C., J.-D. HUANG, C.-C. YEN, C.-H. SHIH und J.-Y. JOU: *Formal Compliance Verification of Interface Protocols*. S. 12–15, 2005.

Betreute studentische Arbeiten

[Bil05] BILGE, HÜSEYIN: *Design and Implementation of a hierarchical Bus System for the reconfigurable XPP Architecture*. Diplomarbeit, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung, 2005.

[Che06] CHEN, BO: *Entwicklung eines Steuerungsystems für ein Network-on-Chip (NoC) und dessen Integration in ein Interface zur Anbindung von Recheneinheiten an das NoC*. Diplomarbeit, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung, 2006.

[Che07] CHEN, CHANG: *Entwicklung einer Steuerungseinheit für ein Network-on-Chip (NoC) und deren Integration in ein Interface zur Anbindung von Recheneinheiten an ein NoC*. Diplomarbeit, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung, 2007.

[Gan05] GANGATHARAN, RAMRASAD: *Development of Data Sources and Sinks for testing of Networks on Chips*. Diplomarbeit, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung, 2005.

[Gro12] GROENER, RODRIGO: *Ansatz zur Virtualisierung der formalen Verifikation serieller Protokolle am Beispiel des LIN-Busses; Approach for the virtualization of formal verification of serial protocols exemplified at the LIN bus*. Diplomarbeit, Karlsruher Institut für Technologie, Institut für Technik der Informationsverarbeitung, 2012.

[Gua03] GUAN, YUWEN: *Entwicklung einer AHB-Speicher Bridge und Analyse verlustleistungsoptimierter System-on-Chip Speichertopologien anhand eines MPEG-4 Dekoders*. Diplomarbeit, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung, 2003.

[Kas05] KASAM, SRINIVAS: *Design, Implementation and Verification of a RAM-based Controller Structure for a Network on Chip*. Diplomarbeit, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung, 2005.

[Kon03] KONRADI, EDUARD: *Hierarchische Standardzellensynthese und Verlustleistungsoptimierung einer dynamisch rekonfigurierbaren Recheneinheit der XPP-Architektur*. Diplomarbeit, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung, 2003.

[Kon05] KONRADI, EDUARD: *Konzeption und Implementierung eines parametrisierbaren Packet-Switched-Routers für Networks-on-Chip*. Diplomarbeit, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung, 2005.

[Krö06] KRÖNER, HERMAN: *Entwicklung eines Interfaces zur Anbindung von Recheneinheiten an ein leitungsgeschaltetes Network-on-Chip*. Diplomarbeit, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung, 2006.

[Qu06] QU, YUANNING: *Erweiterung eines Network-on-Chip-Routers um leitungsgeschaltete Kommunikation*. Diplomarbeit, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung, 2006.

[Rie06] RIES, FLORIAN: *Implementierung des Advanced Audio Codecs (AAC) des DRM (Digital Radio Mondiale) Standards auf der OSYRES Middleware für Multiprozessor Systems-on-Chip Lösungen*. Diplomarbeit, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung, 2006.

[Sch03] SCHMIDT, SEBASTIAN: *Synthese, Test und Verlustleistungsoptimierung eines rekonfigurierbaren System-on-Chip mit integrierter XPP- und LEON-RISC-Architektur*. Diplomarbeit, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung, 2003.

[Son03] SONG, GUODONG: *Synthese, Layoutoptimierung und Anwendungsanalyse eines System-on-Chip mit dem Leon Prozessor und der rekonfi-*

gurierbaren XPP-Architektur. Diplomarbeit, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung, 2003.

[Tan06] TANTRA, FELIKS: *Implementierung eines universellen Router-Modells für Networks-on-Chip in SystemC*. Diplomarbeit, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung, 2006.

[Tan07] TANTRA, FELIKS: *Entwicklung eines automatischen Testsystems für einen DAB/DMB Empfänger*. Diplomarbeit, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung, 2007.

[Tra05] TRAUTMANN, MARTIN: *Entwicklung und Realisierung eines flexiblen Interfaces zur Anbindung von Recheneinheiten an ein Network-on-Chip*. Diplomarbeit, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung, 2005.

[Wan03] WANG, KE: *Synthese, Layoutoptimierung und Anwendungsanalyse eines System-on-Chip mit dem Leon Prozessor und der rekonfigurierbaren XPP-Architektur*. Diplomarbeit, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung, 2003.

[Wan06] WANG, BO: *Implementierung und Test eines NoC auf der RP-Plattform COMPASS*. Diplomarbeit, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung, 2006.

[Wan07] WANG, JING: *Konzeption und Implementierung eines Network-on-Chip Routers zur leitungsgeschalteten Kommunikation*. Diplomarbeit, Universität Karlsruhe, Institut für Technik der Informationsverarbeitung, 2007.

Veröffentlichungen

Konferenzbeiträge

[BBBMG06] BECKER, J. E., C. BIESER, J. BECKER und K. D. MUELLER-GLASER: *Evaluation of a Packet Switching Algorithm for Network on Chip Topologies using a Xilinx Virtex-II FPGA based Rapid Prototyping System*. IEEE International Symposium on Industrial Electronics 2006, 4:3184–3189, July 2006.

[BBT+03] BECKER, J. E., C. BIESER, A. THOMAS, K. D. MUELLER-GLASER und J. BECKER: *Hardware/software co-training by FPGA/ASIC synthesis and programming of a RISC microprocessor-core*. IEEE International Conference on Microelectronic Systems Education, 2003. Proceedings., Seiten 134–135, June 2003.

[BGT+15] BEUTH, T., T. GAEDEKE, C. TRADOWSKY, J. E. BECKER, A. KLIMM und O. SANDER: *The Road to ÏTIV Labs an Integrated Concept for Project-Oriented Systems Engineering Education*. International Journal of Information and Education Technology, 5(4):250–254, 2015.

[BSK+11] BECKER, J. E., O. SANDER, A. KLIMM, S. BULACH, K. WEINBERGER und J. BECKER: *Towards Provable Protocol Conformance of*

Serial Automotive Communication IP. Design & Verification Conference & Exhibition (DVCon), März 2011.

[SKB+09] SANDER, O., A. KLIMM, J. E. BECKER, J. BECKER, T. KIMMESKAMP, J. FORMANN, K. ECHTLE, K. WEINBERGER und S. BULACH: *Sicherung von Zuverlässigkeit und Interoperabilität bei der fahrzeuginternen Kommunikation mittels formaler Verifikation*. VDI-Konferenz Elektronik im Kraftfahrzeug 2009, 2009. 23444.

[TBB04] THOMAS, A., J. E. BECKER und J. BECKER: *Anwendungsspezifische IP-Generierung für zukünftige SoC-Implementierungen in mobilen Kommunikationssystemen*. Dresdner Arbeitstagung Schaltungs- und Systementwurf (DASS ´2004), April 2004.

[WSB05] WOLKOTTE, P. T., G. J. M. SMIT und J. E. BECKER: *Energy efficient NoC for best effort communication*. International Conference on Field Programmable Logic and Applications, 2005., Seiten 197–202, Aug. 2005.

[WSK+05] WOLKOTTE, P. T., G. J. M. SMIT, N. KAVALDJIEV, J. E. BECKER und J. BECKER: *Energy Model of Networks-on-Chip and a Bus*. International Symposium on System-on-Chip, 2005. Proceedings., Seiten 82–85, Nov. 2005.