technische universität
dortmund

Bachelorarbeit

Beispielhafte Entwicklung einer Hardwareproduktlinie anhand einer UART-Familie

Jan Michael Dworschak
23. August 2011

Betreuer:
Prof. Dr.-Ing. Olaf Spinczyk
Dipl.-Inf. Matthias Meier

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Informatik 12
Arbeitsgruppe Eingebettete Systemsoftware
http://ess.cs.tu-dortmund.de

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 23. August 2011

Jan Michael Dworschak

Zusammenfassung

Diese Arbeit beschäftigt sich mit der Entwicklung einer hoch-konfigurierbaren UART-Produktfamilie im Kontext des LavA-Projekts der TU Dortmund. Die LavA-Plattform bietet einen Workflow zur anwendungsspezifischen Generierung von MPSoCs (Multiprocessor Systems on Chip). Ziel der Arbeit ist es zunächst, die aktuell vorhandene, statische UART-Komponente um umfangreiche Konfigurationsmöglichkeiten zu erweitern. Weiterhin wird exemplarisch versucht, XVCL aus dem Workflow zu eliminieren und die Codegenerierung ausschließlich mit Hilfe von Xpand-Templates zu durchzuführen.

Im ersten Teil der Arbeit werden im Rahmen einer Domänenanalyse verschiedene UART-Lösungen im Hinblick auf ihren Funktionsumfang betrachtet. Anhand der hier gewonnenen Erkenntnisse wird im Folgenden der Funktionsumfang für den konfigurierbaren LavA-UART abgesteckt und eine Spezifikation erstellt, auf deren Grundlage der UART dann in VHDL implementiert werden soll. Der zweite Teil der Arbeit behandelt die letztendliche Implementation der entworfenen Komponenten in VHDL und das Einbringen von Konfigurationsmöglichkeiten durch den Einsatz von Xpand. Des Weiteren wird eine Bibliothek von Ansteuerungsfunktionen entwickelt, um den UART an das Betriebssystem zu binden und somit in LavA zu integrieren. Im letzten Teil wird die Arbeit mit einer Evaluation der Umsetzung abgeschlossen. Zum einen soll der umgesetzte UART bezüglich seiner Ressourceneffizienz analysiert werden. Dazu wird in erster Linie untersucht, inwiefern der Umfang der gewählten Konfigurationsmöglichkeiten mit der Ressourcenausnutzung auf dem Ziel-FPGA skaliert. Zum anderen soll untersucht werden, wie gut sich Xpand als Alternative zu XVCL eignet. Es wird betrachtet wo Probleme aufgetreten sind und an welchen Stellen klare Vorteile zu XVCL bestehen.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Motivation . . . 1
- 1.2 Ziele der Arbeit . . . 2
- 1.3 Gliederung der Arbeit . . . 2

2 Grundlagen **5**
- 2.1 Hardwareproduktlinien . . . 5
- 2.2 Funktionsweise und Anwendung eines UART . . . 6
- 2.3 XVCL – XML-based Variant Configuration Language . . . 6
- 2.4 Das LavA-Projekt . . . 8
  - 2.4.1 Hardwarekonfiguration in LavA . . . 8
  - 2.4.2 Der LavA-UART . . . 10
- 2.5 Merkmalmodelle . . . 11

3 Verwendete Technologien **13**
- 3.1 Entwicklung von Templates mit Xpand . . . 13
  - 3.1.1 Die Expressions-Subsprache . . . 13
  - 3.1.2 Struktur von Xpand-Dateien . . . 14
  - 3.1.3 Erweiterungen durch Xtend . . . 15
  - 3.1.4 Formulierung von Modellconstraints mit Check . . . 15
  - 3.1.5 Workflowdateien als Verknüpfung der Sprachen . . . 16
- 3.2 Konfigurationsmittel in VHDL . . . 18

4 Analyse **19**
- 4.1 UARTs gängiger Mikrocontroller . . . 19
  - 4.1.1 Atmel ATmega8/16/32 . . . 19
  - 4.1.2 NXP LPC2377/78 . . . 20
  - 4.1.3 Microchip PIC32 . . . 21
- 4.2 IP-Core UARTs . . . 22
  - 4.2.1 UART16750 . . . 22
  - 4.2.2 UART 16550 core . . . 23
- 4.3 Zusammenfassung und Merkmalmodell . . . 24

5 Entwurf des UART **29**
- 5.1 Funktionsumfang des UART . . . 29
  - 5.1.1 Formatierung und Kommunikation . . . 30

5.1.2 Interrupts . . . 30
5.1.3 Pufferung . . . 30
5.2 Umsetzung der Konfigurationspunkte . . . 31
5.3 Eclipse-Metamodell . . . 31
5.4 Entwurf einzelner Komponenten . . . 32
5.4.1 RX- und TX-Einheit . . . 33
5.4.2 Steuereinheit . . . 33
5.4.3 Anbindung an den Wishbone-Bus . . . 34
5.4.4 Blockdiagramm des UART . . . 35

6 Implementation **37**
6.1 Implementation einzelner Komponenten in VHDL . . . 37
6.1.1 Die Top-Level-Entität . . . 37
6.1.2 RX-Einheit . . . 38
6.1.3 TX-Einheit . . . 39
6.1.4 FIFO-Puffer . . . 40
6.1.5 Steuereinheit . . . 41
6.1.6 Baudratengenerator . . . 42
6.1.7 Konfigurationsregister . . . 42
6.1.8 Paritätsprüfung und -generierung . . . 42
6.2 Die Codegenerierung . . . 43
6.2.1 Constraints . . . 43
6.2.2 Struktur der Templates . . . 44
6.2.3 Extensions . . . 46
6.2.4 Der Workflow . . . 46
6.3 Ansteuerung des UART . . . 47
6.3.1 Senden von Daten . . . 47
6.3.2 Auslesen empfangener Daten . . . 47
6.3.3 Konfiguration . . . 48
6.3.4 Interruptbehandlung . . . 48

7 Evaluation **51**
7.1 Methodik des UART-Tests . . . 51
7.1.1 Auswahl der Testsysteme . . . 52
7.1.2 Ablauf des Laufzeittests . . . 52
7.2 Evaluation des UART . . . 53
7.2.1 Minimalkonfiguration . . . 53
7.2.2 Teilmenge der möglichen Komponenten . . . 54
7.2.3 Kompletter Funktionsumfang . . . 54
7.2.4 Auswertung der Ergebnisse . . . 55
7.3 Evaluation der Codegenerierung . . . 55
7.3.1 Vergleich von Xpand und XVCL . . . 55
7.3.2 Eignung von Xpand . . . 57

8 Fazit und Ausblick **59**
8.1 Zusammenfassung 59
8.2 Ausblick 60

Literaturverzeichnis **63**

Abbildungsverzeichnis **65**

Tabellenverzeichnis **67**

1 Einleitung

Eingebettete Systeme nehmen heutzutage einen Großteil der auf dem Markt befindlichen Rechnersysteme ein. In nahezu jedem elektrischen Gerät des Alltags findet man eingebettete Rechnerhardware. Gerade in technisch anspruchsvollen und rechenintensiven Anwendungen, wie Multimediageräten, Telekommunikation oder auch Verschlüsselungssystemen für Netzwerke, findet man dabei besonders sogenannte MPSoCs (Multiprocessor Systems on Chip). Ein MPSoC vereint mehrere Prozessoren, Speicher und verschiedenste benötigte Peripheriegeräte auf einem Chip mit dem Ziel die jeweilige Anwendung möglichst schnell und effizient auszuführen. Um dieses Ziel zu erreichen, ist es nötig die Hardware auf den Anwendungszweck maßzuschneidern. Eine Möglichkeit dies zu tun ist der Entwurf eines ASIC (Application-Specific Integrated Circuit), also einer hochspezifischen, aber statischen Hardware. Da sich das Design eines ASIC aus Kostengründen oft nur für Schaltungen lohnt, die in großen Stückzahlen produziert werden [1], werden heute viele eingebettete Systeme mithilfe von Hardwarebeschreibungssprachen, wie VHDL, Verilog oder SystemC, und rekonfigurierbarer Logik, in Form von FPGAs (Field-Programmable Gate Arrays), entworfen. Das System wird dabei in einer der genannten Beschreibungssprachen textuell beschrieben und dann auf einer generischen Hardwareplattform ausgeführt. Diese Technik bringt zum einen den Vorteil mit sich, auf derselben physikalischen Hardware verschiedenste Hardwarelösungen realisieren zu können und so Kosten zu minimieren. Zum anderen wird es möglich lang bekannte Methoden und Techniken aus der Softwareentwicklung für den Hardwareentwurf zu erschließen, da sich durch den Einsatz von rekonfigurierbaren Logikbausteinen und Hardwarebeschreibungssprachen die Entwurfsmodelle immer mehr annähern.

1.1 Motivation

Das LavA-Projekt der TU Dortmund [2], das den Kontext dieser Bachelorarbeit bietet, verfolgt einen solchen Ansatz. Es verwendet das Konzept der modellgetriebenen Softwareentwicklung (MDSD) [3], um ausgehend von einem abstrakten Modell, maßgeschneiderte Hardware zu entwickeln. Die LavA-Plattform bietet bereits das feingranular konfigurierbare Betriebssystem *CiAO* [4] sowie eine Vielzahl von konfigurierbaren Hardwarekomponenten, geschrieben in VHDL. Dazu zählen verschiedene Prozessoren und IPC[1]-Komponenten, aber auch Peripheriekomponenten wie UARTs. Ein UART ist ein Bauelement, das die Schnittstelle zwischen einem parallelen Datenbus und einer seriellen Leitung ermöglicht, sodass ein Mikrocontroller mit einem anderen Rechner oder Modem

[1] Interprozesskommunikation

über Standards wie RS–232 kommunizieren kann [5]. Um hohe Flexibilität im Entwurf und die größtmögliche Spezifität in der zur entwerfenden Hardware zu gewährleisten, ist es essentiell, die jeweiligen Einzelkomponenten konfigurierbar und flexibel zu halten. Da die aktuell in LavA integrierte UART-Komponente diesen Anspruch noch nicht erfüllen kann, soll sie in dieser Arbeit erweitert werden. Um mit der LavA-Plattform ein MP-SoC zu entwickeln, muss zunächst auf der Grundlage eines Metamodells ein konkretes Modell der Hardware entworfen werden. Dieses Modell wird mit Hilfe der Templatesprache Xpand [6] [7] in eine Spezifikation für die XML-basierte Konfigurationssprache XVCL [8] transformiert, aus der dann der letztendliche Code erzeugt wird. Besonders bei komplexeren Entwürfen zeigt sich hier allerdings ein Problem. Die Lesbarkeit der XML-Dokumente, die für die Konfiguration über XVCL erzeugt werden müssen, ist aufgrund ihrer Größe und Komplexität stark eingeschränkt. Dies hat einen nachhaltig negativen Einfluss auf die Wartbarkeit des Systems. Die Neuentwicklung einer Komponente bietet sich dabei an, um konzeptionell zu überprüfen, inwiefern XVCL aus dem Workflow herausgelöst und komplett durch Xpand ersetzt werden kann.

1.2 Ziele der Arbeit

Ziel dieser Arbeit ist es zunächst, ausgehend von der aktuell in LavA vorhandenen statischen UART-Komponente, einen hoch-konfigurierbaren UART in VHDL zu entwickeln. Zu diesem Zweck wird anfangs eine Domänenanalyse durchgeführt, um den späteren Funktionsumfang des UART abzustecken. Auf der Grundlage dieser Analyse soll dann ein Blockdiagramm entstehen, nach dem der UART in VHDL implementiert werden kann. Um den UART in LavA zu integrieren, soll daraufhin eine Bibliothek von Funktionen für das Betriebssystem implementiert werden, die es ermöglicht, die neuen Features, hauptsächlich die Konfiguration im Betrieb, auszunutzen. Bezüglich der Konfigurationsmöglichkeiten wird anstelle der sonst in LavA verwendeten Konfiguration über XVCL, soweit möglich, ausschließlich Xpand verwendet. Das so entstandene System soll abschließend nach Skalierbarkeit der VHDL-Komponente und Lesbarkeit bzw. Komplexität der Xpand-Templates evaluiert werden.

1.3 Gliederung der Arbeit

Im zweiten Kapitel werden zunächst einige im Kontext dieser Arbeit wichtige Grundlagen der modellgetriebenen Hardwareentwicklung betrachtet. Darunter finden sich das LavA-Projekt der TU Dortmund, aber vor allem auch das Konzept der Hardwareproduktlinien und die Templatesprache XVCL. Das dritte Kapitel befasst sich im Anschluss mit Technologien, die im Laufe der Arbeit zur Entwicklung der UART-Familie verwendet werden. Dazu zählen die Templatesprache Xpand und die verwandten Sprachen Xtend und Check sowie Konfigurationsmittel von VHDL. Das vierte Kapitel beschreibt die grundlegende Domänenanalyse, die die Basis für Entwurf und Implementation des UART bildet. Das fünfte Kapitel erläutert den Entwurfsprozess und die Spezifikation der

zu entwickelnden UART-Komponente. Kapitel 6 zeigt die wesentlichen Teile der fertigen Komponente und erläutert den Implementationsprozess sowie die Schnittstelle zum Betriebssystem. Abschließend wird in Kapitel 7 eine Evaluation des entstandenen UART vorgenommen. Der erste Teil des Kapitels erläutert die Methodik, mit der der UART nach Abschluss der Implementierung besonders bezüglich der verschiedenen Konfigurationsmöglichkeiten getestet wurde. Im zweiten Teil des Kapitels wird dann betrachtet, wie sich der entwickelte UART bezüglich der Ressourcenausnutzung auf dem Ziel-FPGA verhält. Der dritte und letzte Teil des Kapitels befasst sich schließlich mit Xpand und der Eignung als Ersatz für die bisher verwendete XVCL-Lösung. Kapitel 8 schließt die Arbeit mit einem Fazit und Ausblick ab.

2 Grundlagen

In diesem Kapitel sollen, bevor die eigentliche Entwicklung der UART-Familie thematisiert wird, grundsätzliche Konzepte der modellgetriebenen Hardwareentwicklung, besonders mit Bezug auf das LavA-Projekt, näher erläutert werden, um das weitere Verständnis zu erleichtern. Zu Beginn werden das Konzept von Hardwareproduktlinien sowie die Funktionsweise und das Anwendungsfeld eines UART genauer erläutert. Im Folgenden wird dann näher auf die Funktionsweise der LavA-Plattform eingegangen. Zu diesem Zweck werden zunächst die wichtigsten Konzepte und Technologien, die im LavA-Workflow verwendet werden, erläutert. Dazu zählen XVCL, Xpand und Möglichkeiten der Konfiguration innerhalb von VHDL. Detailliertere Erläuterungen zu den letzten beiden Punkten finden sich hierbei insbesondere in Kapitel 3. Abschließend werden Merkmalmodelle thematisiert, die im Kontext der Domänenanalyse eine wichtige Rolle spielen.

2.1 Hardwareproduktlinien

Der Begriff der Produktlinien stammt ursprünglich aus der Softwareentwicklung. Mit der zunehmenden Verbreitung von rekonfigurierbarer Logik in Form von FPGAs und Popularität von Hardwarebeschreibungssprachen, wie VHDL, Verilog oder SystemC, wird dieses Konzept allerdings auch auf Hardwareentwicklungsprozesse anwendbar. Böckle et al. [9] beschreiben die Produktlinienentwicklung als ein Konzept, dass auf „organisierter Wiederverwendung und organisierter Variabilität“ basiert. Die Grundlage der Produktlinien bildet die Zuordnung von Software zu bestimmten Problemdomänen. Czarnecki und Eiesenecker [10] beschreiben eine Domäne als einen abgeschlossenen Raum von Wissen, Technologien und Konzepten, definiert durch die Interessengruppen (Stakeholder) dieses Gebietes. Insbesondere ist darin auch das Wissen um die Entwicklung entsprechender domänenspezifischer Software eingeschlossen, sodass eine Domäne aus technischer Sicht auch als eine Menge von Systemen für einen bestimmten Problembereich betrachtet werden kann. Ausgehend von dieser Domänensicht wird nun versucht, Software nicht für jeden Anwendungszweck von Grund auf neu zu entwickeln, sondern auszunutzen, dass Anwendungen aus einer bestimmten Problemdomäne oft viele gemeinsame Strukturen und Funktionalitäten besitzen, aber nur in einzelnen Merkmalen variieren. Dies kann ausgenutzt werden, indem man einen gemeinsamen Kern von wiederverwendbaren Komponenten innerhalb einer Entwicklungsplattform bereitstellt. Ausgehend von dieser Menge an Artefakten werden dann anwendungsspezifische Varianten der Softwarelösung produziert. Einerseits bringt dies den Vorteil mit sich, dass die Software–Artefakte innerhalb der Entwicklungsplattform meist ausgereifter sind, als völlig neu entwickelte

Lösungen. Andererseits erhofft man sich, trotz des initialen Mehraufwandes, eine insgesamt kürzere Entwicklungszeit pro Produkt. Um einen solchen Ansatz erfolgreich umzusetzen ist es wichtig, eine gute Beschreibung der gemeinsamen Eigenschaften, aber auch der jeweiligen Variationspunkte einer Produktlinie, zu erstellen. Zu diesem Zweck wird eine Domänenanalyse durchgeführt, deren Ergebnis in einem Domänenmodell formuliert wird. Genauere Erläuterungen zu Domänen– und Merkmalmodellen finden sich im entsprechenden Abschnitt dieses Kapitels.

2.2 Funktionsweise und Anwendung eines UART

Das Kürzel *UART* steht für „Universal Asynchronous Receiver/Transmitter“ [5]. Ein UART hat die Aufgabe, Daten, die über eine Bus–Anbindung parallel eingegeben werden, in serielle Ausgabesignale (beispielsweise RS-232) zu übersetzen und umgekehrt. Die Bezeichnung *universal* soll dabei andeuten, dass die eigentlichen Kommunikationsparameter, wie Baudrate und Signalpegel der Ein– und Ausgabe nicht festgelegt, sondern konfigurierbar sind. Da jeder der Kommunikationspartner im Normalfall seine eigene Taktgebung hat, wird bei der seriellen Kommunikation ein Rahmen von Start– und Stoppbits genutzt, um Datenwörter zu kodieren. Häufige Varianten des UARTs sind der sogenannte *Dual UART* oder *DUART*, der zwei UARTs in einem Bauteil vereint, oder auch sogenannte *USARTs*, die neben dem klassischen asynchronen Kommunikationsprinzip zusätzlich synchrone Kommunikation anbieten. Im Fall der synchronen Kommunikation werden die Daten nicht in einem festen Rahmen mit Start– und Stoppbits eingebettet, sondern über ein mitgeliefertes Taktsignal synchronisiert. Anders als bei der asynchronen Kommunikation, bei der zwischen Stoppbit eines ersten und Startbit eines nächsten Datenwortes theoretisch eine beliebig lange Pause liegen kann, folgen die Datenwörter bei der synchronen Kommunikation in einem stetigen Strom aufeinander. UARTs finden sich heutzutage in fast allen Mikrocontrollern. Ein weit verbreitetes und de facto Standardmodell ist der 16550 UART der National Semiconductor Corporation [11].

2.3 XVCL – XML-based Variant Configuration Language

Bei XVCL [8] handelt es sich um eine XML-basierte Markup-Sprache, die sich besonders an Entwickler richtet, die mit Produktfamilien oder im Allgemeinen mit Variationen von Software arbeiten. Die Grundelemente von XVCL sind die sogenannten *x-frames*. Ein x-frame ist eine XML-Datei, die beliebigen Text – im Kontext von LavA handelt es sich dabei um VHDL-Code – und Anweisungs-Tags enthält. Bezüglich der Anweisungen unterstützt XVCL eine Reihe von Konstrukten, die aus den meisten Programmiersprachen bekannt sind. Es gibt Variablendeklarationen für skalare Variablen und Listen von Werten, eine `select`–Anweisung und `while`-Schleifen. Mithilfe dieser Werkzeuge lassen sich komplexeste Konstrukte erzeugen. Innerhalb eines x-frames können

durch Verwendung des *<adapt>*-Tags andere x-frames eingebunden werden. Dies bietet die Möglichkeit der Einkapselung und Wiederverwendung von Komponenten. Durch *<adapt>* entsteht bei Verflechtung mehrerer x-frames eine baumartige Hierarchie. Die Wurzel dieser Hierarchie und damit den Einstiegspunkt des XVCL-Prozessors stellt der *Specification Frame*, kurz SPC, dar. Eine zusammenhängende Menge von x-frames wird als x-framework bezeichnet. Um Daten in einen adaptierten x-frame einzubringen gibt es zwei Möglichkeiten. Zum einen sind Variablen eines übergeordneten x-frames auch im untergeordneten sichtbar, zum anderen können mit *<break>* Breakpoints definiert werden, an denen dann Text aus dem übergeordneten x-frame eingefügt werden kann. Dies geschieht über *<insert>*. Ein Beispiel für ein x-framework mit zwei x-frames ist in Abbildung 2.1 dargestellt. In der Abbildung links oben ist der SPC des x-frameworks zu

```
<x-frame name="SPC" outfile="MPSoC.vhd">

    <set var="UART" value="1"/>

    <adapt x-frame="MPSoC">
        <insert break="SIGNALS">
            <select option="UART">
                <option value="1">
        uart_rx      : in  std_logic;
        uart_tx      : out std_logic;
                </option>
            </select>
        </insert>
    </adapt>
</x-frame>
```

```
<x-frame name="MPSoC">

entity MPSoC is
 port(
   clock : in  std_logic;
   reset : in  std_logic;
  <break name="SIGNALS"/>
 );
end MPSoC;

</x-frame>
```

```
entity MPSoC is
    port(
        clock       : in  std_logic;
        reset       : in  std_logic;
        uart_rx     : in  std_logic;
        uart_tx     : out std_logic;
    );
end MPSoC;
```

Abbildung 2.1: Beispiel für ein x-framework [12]

sehen. Im SPC wird eine Variable *UART* definiert und mit 1 initialisiert. Der x-frame auf der rechten Seite wird eingebunden und das Resultat der `select`–Anweisung an den Breakpoint *SIGNALS* dieses x-frames gebunden. Durch die Belegung der Variable mit 1 resultiert dann der im unteren Abschnitt dargestellte Quellcode. Wäre die Variable mit 0 oder einem anderen Wert belegt, so würde innerhalb des *<select>*–Kontextes kein Code ausgegeben und der Code würde nur die ersten beiden Signaldeklarationen enthalten. Eine vollständige Dokumentation der Funktionen und Sprachelemente ist unter [13] zu finden.

2.4 Das LavA-Projekt

Das LavA-Projekt zielt auf das Co–Design bzw. die Co–Konfiguration der Hardwarebeschreibung eines Systems und passender Betriebssystemsoftware ab. Da die im Verlauf dieser Arbeit zu entwickelnden Komponenten Teil der Hardwareschicht der LavA-Plattform sind und eine Weiterentwicklung des Hardware–Konfigurationsprozesses untersucht wird, soll das Augenmerk hier hauptsächlich auf diesem Teil des LavA-Workflows liegen. Es sei aber angemerkt, dass es sich hierbei keinesfalls um den Gesamtumfang der LavA-Plattform handelt. Wie bereits im vorangehenden Kapitel angedeutet, handelt es sich bei LavA um eine Plattform zur anwendungsspezifischen Maßschneiderung von MPSoCs. Besonders im Vordergrund steht dabei, die Komplexität und Fehleranfälligkeit des typischen Entwicklungsprozesses zu reduzieren. LavA erreicht dies durch Anwendung zweier zentraler Konzepte der Softwareentwicklung. Das sind zum einen ein modellgetriebener Entwicklungsansatz unter Verwendung diverser Elemente aus dem *Eclipse Modeling Framework* (EMF) [14] und zum anderen die gezielte Wiederverwendung von Hardware– und Softwarekomponenten [12].

2.4.1 Hardwarekonfiguration in LavA

Der Ausgangspunkt eines neuen Entwurfs in LavA ist das Metamodell. Das Metamodell ist in Abbildung 2.2 dargestellt. Es beschreibt die grundsätzliche Architektur, die jedem mit LavA entwickelten Hardwaresystem zugrunde liegt. Ein MPSoC besteht demnach

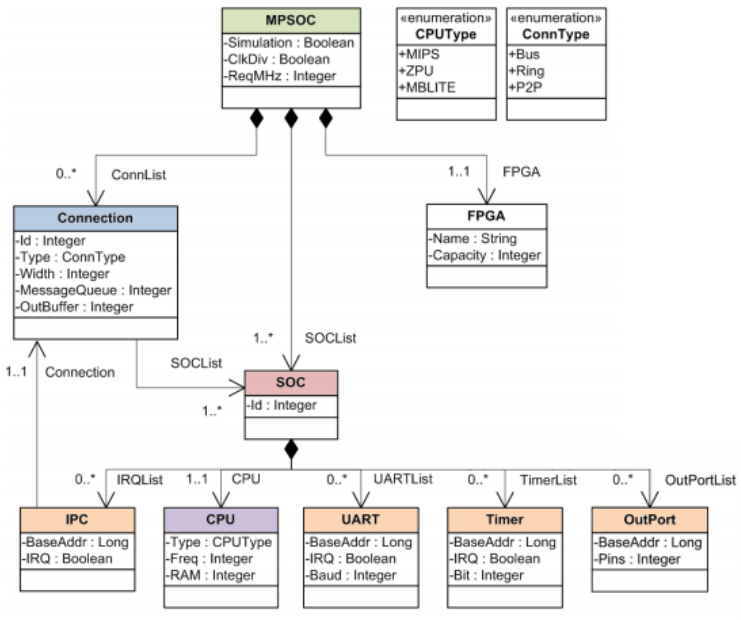

Abbildung 2.2: Das Hardware-Metamodell in LavA

aus einer gewissen Anzahl von SoCs und entsprechenden Connections zwischen diesen. Ein SoC wiederum besteht aus einem Prozessor (CPU) und weiteren Peripheriegeräten, wie UARTs, Timern und Interprozesskommunikationsgeräten. Des Weiteren gibt es zu jedem MPSoC auch schon ein Ziel-FPGA. Zusätzlich zu den in der Abbildung ersichtlichen Abhängigkeiten und Zusammenhängen, gibt es noch weitere Constraints, die mit *Check* [15] formuliert sind. Bei Check handelt es sich um eine Constraintsprache, ähnlich wie *OCL*[1], mit der man zusätzliche Beschränkungen und Abhängigkeiten in einem konkreten Modell annotieren kann. Sie unterstützt die automatische Validierung eines Modells bezüglich der Constraints. Auf der Basis dieses Metamodells muss, um ein Sys-

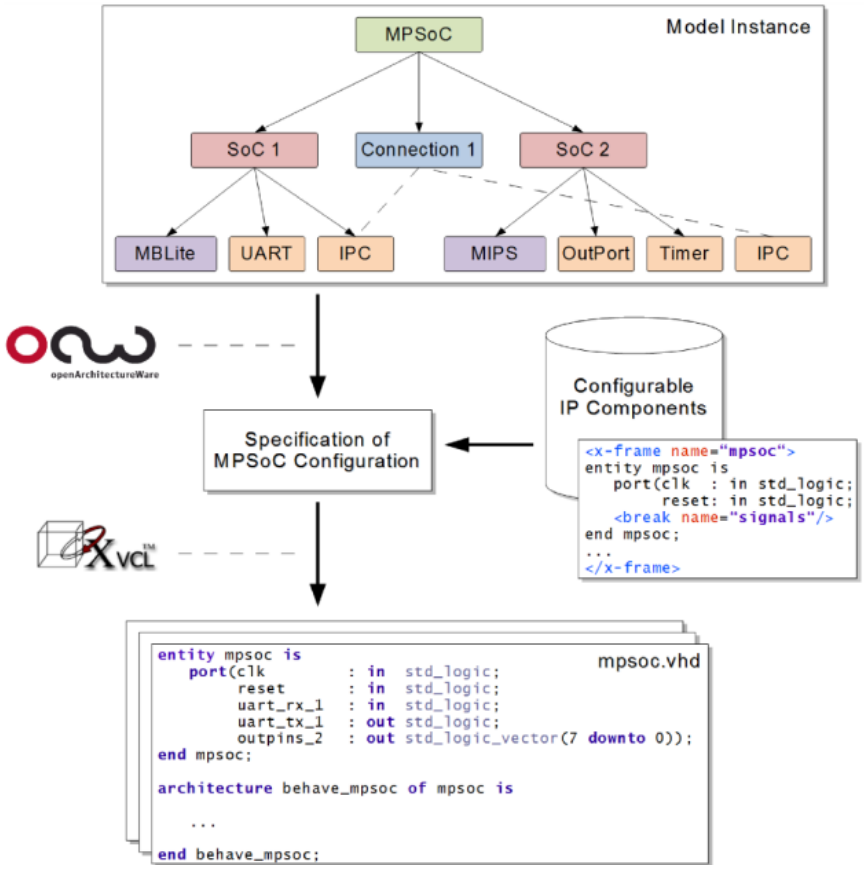

Abbildung 2.3: Die Hardware-Konfiguration der LavA-Plattform [2]

tem zu modellieren, ein konkretes Modell abgeleitet werden, in das die Anforderungen des Anwendungsgebietes einfließen. Dies geschieht mit Hilfe eines grafischen Editors. Nachdem mit der Erstellung des konkreten Modells die Auswahl und Konfiguration der einzelnen Komponenten vorgenommen wurde, müssen die gewählten Optionen mit zu generierenden Quellcode in Verbindung gesetzt werden. LavA verwendet zur Codegenerierung XVCL. Daher wird ein Xpand-Template verwendet, um die Werte aus dem vom

[1]Object Constraint Language

Benutzer erzeugten MPSoC-Modell zu extrahieren und daraus einen SPC zu erzeugen. Die VHDL-Komponenten, die als XVCL x-frames bzw. x-frameworks vorliegen, können dann leicht vom SPC adaptiert und parametrisiert werden. So entsteht ein zusammenhängendes System aus VHDL-Dateien, welches dann auf ein FPGA synthetisiert werden kann. Eine Visualisierung dieses Ablaufs ist in Abbildung 2.3 zu sehen.

2.4.2 Der LavA-UART

Dieser Abschnitt soll einen kurzen Überblick bieten, wie der Ausgangszustand der UART-Komponente im LavA-Projekt ist. Die Architektur des UART ist schematisch in Abbildung 2.4 zu sehen. Die drei wesentlichen Komponenten des UART sind die Steuerein-

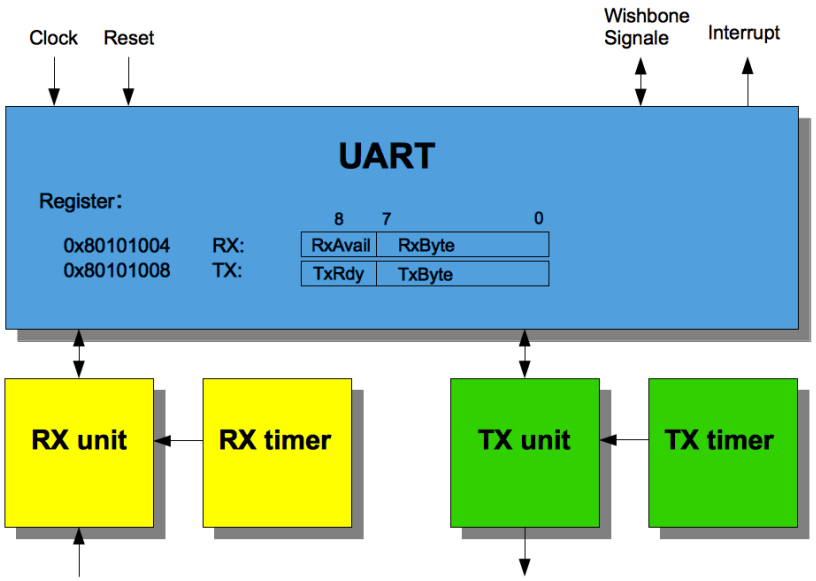

Abbildung 2.4: Der Aufbau der ursprünglichen LavA UART-Komponente

heit (in der Abbildung blau), die Empfängereinheit (gelb) und die Sendeeinheit (grün). Die Zeitgebung der Sende– und Empfangseinheiten werden durch zwei Timer geregelt, die Baudrate ist dabei allerdings statisch und nur zur Erzeugungszeit der VHDL-Komponente wählbar. Die parallele Dateneingabe und das Auslesen der empfangenen Daten werden über ein Wishbone–Interface [16] durchgeführt. Die interne Speicherung der Daten erfolgt über zwei Register mit einer Breite von je 9 Bit, darunter 8 Datenbits und 1 Statusbit. Da es keine Pufferregister zum Senden und Empfangen gibt, kann es passieren, dass Daten schneller in die Register geschrieben werden, als sie verarbeitet werden, wodurch es leicht zu Datenverlusten kommt. Der UART verfügt über eine Interrupt-Leitung, über die signalisiert wird, wenn Daten zur Abholung bereit sind.

2.5 Merkmalmodelle

Merkmalmodelle [17] stellen neben Abgrenzungen der Domäne nach außen, Definitionen von speziellem Vokabular und Konzeptmodellen (UML-Klassendiagramme, ER-Diagramme oder ähnliches) einen wichtigen Bestandteil eines Domänenmodells dar. Ein Merkmalmodell selbst besteht aus einem Merkmaldiagramm in Verbindung mit textuellen Beschreibungen und weiteren Einschränkungen oder Implikationen beispielsweise zur Erläuterung einer Widersprüchlichkeit zwischen zwei Knoten eines Diagramms. Abbildung 2.5 zeigt ein Beispiel für ein Merkmaldiagramm. Syntaktisch stellt ein Merkmaldia-

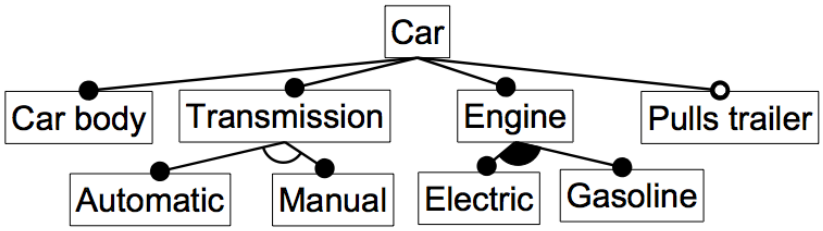

Abbildung 2.5: Beispiel eines Merkmaldiagramms. Quelle [18]

gramm einen gerichteten azyklischen Graphen dar, dessen Wurzel die zu beschreibende Produktfamilie, oft auch als *Konzept* bezeichnet, repräsentiert. Die Kinder eines Knotens sind seine Merkmale, dabei deutet eine Kante mit ausgefülltem Kreis am Ende ein verpflichtendes Merkmal und eine Kante mit nicht ausgefülltem Kreis ein optionales Merkmal an. Im Beispiel muss jedes Auto eine Karosserie, ein Getriebe und einen Motor haben, während ein Anhänger optional bleibt. Bögen zwischen zwei Kanten zeigen an, dass die durch den Bogen verbundenen Merkmale eine Gruppe bilden. Ein einfacher Bogen bedeutet, dass die Gruppe eine Reihe von Alternativen darstellt. Jedes Auto braucht ein Getriebe, es kann aber nur entweder Gangschaltung oder Automatik haben. Bei einem ausgefüllten Bogen handelt es sich um eine Gruppe von kumulativen Merkmalen. Es können eines oder mehrere gewählt werden. Bezogen auf das Beispiel kann man also ablesen, dass der Motor eines Autos mindestens ein Benzinmotor oder Elektromotor sein muss. Es könnte sich aber auch um ein Hybridfahrzeug handeln, das beides unterstützt.

3 Verwendete Technologien

3.1 Entwicklung von Templates mit Xpand

Bei Xpand [6][7] handelt es sich um eine Templatesprache, die ursprünglich als Teil des MDSD–Frameworks openArchitectureWare entworfen wurde. Mittlerweile sind die einzelnen Komponenten dieses Frameworks (Xpand, Xtend und Check) größtenteils unabhängige Projekte innerhalb des Eclipse Modeling Frameworks geworden. Die Syntax von Xpand ähnelt bekannten Markup-Sprachen, wie XML, insofern, dass Anweisungen in Tags formuliert und ineinander verschachtelt werden können. Tags werden dabei im Gegensatz zu HTML oder XML nicht durch einfache spitze Klammern, sondern durch die Zeichen « bzw. », sogenannte Guillemets oder französische Anführungszeichen, begrenzt. Die Besonderheit von Xpand liegt darin, dass Templates auf einem zuvor in einem grafischen Editor erstellten Modell basieren. Einzelne Elemente dieses Modells können und müssen innerhalb des Templates direkt referenziert werden. Auf diese Weise können leicht Parameter von der Modellierungs– auf die Quelltextebene übertragen werden.

3.1.1 Die Expressions-Subsprache

Die drei Hauptkomponenten des ehemaligen openArchitectureWare-Frameworks stützen sich syntaktisch auf eine gemeinsame Grundlage. Die Expressions-Subsprache ist syntaktisch eine Mischung aus Java und OCL und dient dazu, oft benötigte Funktionen, wie boole'sche Operationen, Arithmetik und Zugriffe auf Modellelemente durchzuführen. Die grundlegenden Funktionen von Expressions orientieren sich in ersten Linie an Java, was durch das Beispiel in Abb. 3.1 illustriert wird. Komplexere Funktionen, wie

```
myElement.name              // Attributzugriff
"my name is "+name+" !"     // String-Konkatenation
true || false               // Boole'sche Ausdrücke
-47.11 * 42                 // Arithmetik
```

Abbildung 3.1: Beispiel für einfache Expressions

beispielsweise die Selektion einzelner Elemente mit bestimmten Eigenschaften aus einer Liste, orientieren sich hingegen stark an OCL. Anweisungen, wie `select`, `collect` oder `forAll`, sind identisch zu den entsprechenden OCL-Versionen. Die genauere Erläuterung der einzelnen Operationen ist für das weitere Verständnis nicht entscheidend. Deshalb sei an dieser Stelle auf die detaillierte Dokumentation der Expressions [19] bzw. OCL selbst [20] verwiesen.

3.1.2 Struktur von Xpand-Dateien

Templatedateien bestehen strukturell aus drei Arten von Ausdrücken. Zunächst können Modelle, auf die sich die in der Datei definierten Templates bieziehen sollen, importiert werden. Dazu wird in Xpand das `IMPORT`–Statement verwendet. Neben Modellen können auch Xtend-Dateien in Templatedateien eingebunden werden, was über das Schlüsselwort `EXTENSION` geschieht. Auf diese Weise können Bibliotheken von selbst definierten Operationen für die Verwendung innerhalb der jeweiligen Templatedatei sichtbar gemacht werden. Die dritte Art von Statement, die das eigentliche Kernstück einer Templatedatei ausmacht, bilden sogenannte Templatedefinitionen. Eine Templatedefinition wird durch das Schlüsselwort `DEFINE` eingeleitet und besteht aus einem (im jeweiligen Namensraum) eindeutigen Bezeichner und einer optionalen Parameterliste sowie dem Namen des Modellelementes, auf das sich das Template beziehen soll. Templatedefinitionen beinhalten einfache Textelemente und eine beliebige Anzahl von Anweisungen aus dem Sprachumfang von Xpand. Jeder Definitionsblock, im folgenden als Template bezeichnet, kann eine eigene Ausgabedatei spezifizieren. Dies geschieht über das `FILE`–Statement. Jedes Template kann weiterhin über das `EXPAND`–Statement andere Templates einbinden und somit deren Ausgabe an der jeweiligen Stelle im Template einfügen. Das Expandieren von Templates bietet eine Möglichkeit zur Verkapselung und Wiederverwendung von Templates in unterschiedlichen Kontexten. Es ist vergleichbar mit dem *<adapt>*-Tag in XVCL (siehe Abschnitt 2.3). Neben diesen templatespezifischen Befehlen unterstützt Xpand eine Reihe von Anweisungen, die auch aus den meisten anderen Programmiersprachen bekannt sind. Zum einen gibt es `FOR`- und `FOREACH`-Befehle, die in Xpand benutzt werden, um Templates auf Elemente des Modells anzuwenden. Bei der Verwendung von `FOREACH` ist es hierbei wichtig zu beachten, dass das Argument der Anweisung im Modell kein skalarer Datentyp sein darf, sondern ein Collection-Objekt, also eine Liste, Menge oder ähnliches sein muss. Des Weiteren gibt es konditionale Anweisungen in Form des üblichen `IF`-Blocks mit optionaler Alternativanweisung innerhalb eines `ELSE`-Statements und beliebig vielen, dazwischen angeordneten `ELSEIF`-Blöcken. Xpand erlaubt zudem lokale Variablendefinitionen. Mit Hilfe des `LET`-Statements kann ein Block deklariert werden, in dem die entsprechende Variable sichtbar ist. Im Sprachumfang sind weiterhin drei Metaanweisungen enthalten, die nicht direkt in den Generierungsprozess eingreifen, aber dennoch bei der Entwicklung von Templates nützlich sein können. Zum einen bietet Xpand die Möglichkeit innerhalb eines `REM`-Blocks Kommentare in das Template einzubringen. Dies kann gerade bei großen und komplexen Templates nützlich sein, um die Übersichtlichkeit zu verbessern. Die zweite Metaanweisung ist das `ERROR`-Statement. Wird bei der Abarbeitung des Templates eine solche Anweisung erreicht, dann bricht die Bearbeitung mit einer entsprechend angegebenen Fehlermeldung ab. Es ist allerdings anzumerken, dass dieses Feature nicht benutzt werden sollte, um Fehler im Modell abzufangen, da für solche Fälle der Sprachumfang von Check besser geeignet ist und die Trennung von Modellvalidierung und Codegenerierung sonst nicht gegeben wäre. Zuletzt gibt es noch die Möglichkeit, innerhalb eines `PROTECT`-Blocks ein Code- oder Textsegment anzugeben, dass bei wiederholten Generatordurchläufen nicht mehr überschrieben bzw. verändert werden kann. Dies kann nützlich sein, um die ver-

sehentliche Veränderung von handgeschriebenen Codesegmenten zu vermeiden.

3.1.3 Erweiterungen durch Xtend

Wie bereits angedeutet, können Xpand-Templates neben den Standardfunktionen der Sprache auch weitere, selbst definierte Operationen, sogenannte Extensions, beinhalten. Solche selbst definierten Subroutinen werden mit Hilfe der Sprache Xtend in Extension-Dateien mit der Endung `.ext` definiert. Grundsätzlich gibt es zwei Möglichkeiten eine Extension zu formulieren. Einerseits bietet das Framework die Möglichkeit, statische Java-Methoden, die im Modell definiert sein müssen, zu verwenden. Diese Variante ist lediglich eine Abbildung von Funktionen auf der Modellierungsebene auf die Ebene des Codegenerators. Andererseits ist es möglich mit Hilfe von Expressions eigene Funktionen zu erstellen und so ein hohes Maß an Dynamik in den Generierungsprozess einzubringen. Eine Extension besteht aus einem Rückgabetypen, einem Bezeichner, einer (möglicherweise leeren) Parameterliste und der Deklaration der auszuführenden Operationen. Extensions können insbesondere auch rekursiv sein. Dies kann zum Beispiel dann nützlich sein, wenn für ein Objekt der vollständig qualifizierte Name bestimmt werden soll. Eine Extension, die dies leistet, ist in Abbildung 3.2 dargestellt.

```
String fullyQualifiedName(NamedElement n) : n.parent == null ? n.name:
    fullyQualifiedName(n.parent)+'::'+n.name
;
```

Abbildung 3.2: Extension zur Bestimmung des FQDN[1][21]

Solang das als Parameter übergebene Objekt ein Elternobjekt hat, wird hier die Extension mit dem Elternobjekt erneut aufgerufen und der Name des aktuellen Objektes an das Resultat des Aufrufs angehängt. Xtend bietet weiterhin auch Mittel zur Modelltransformation (M2M) und zur aspektorientierten Programmierung im Kontext der Wiederverwendung von proprietären Generatorkomponenten. Da diese Teile des Sprachumfangs im weiteren Verlauf dieser Arbeit jedoch keine Anwendung finden, wird an dieser Stelle nicht im Detail darauf eingegangen.

3.1.4 Formulierung von Modellconstraints mit Check

Die Sprache Check bietet die Möglichkeit neben den impliziten Vorgaben des Metamodells explizite Einschränkungen zu formulieren, die ein Modell einhalten muss, um korrekt zu sein [15]. Solche Constraints werden in eigenen Dateien mit der Endung `.chk` deklariert und können bei der Codegenerierung zur Modellvalidierung verwendet werden. Ein Constraint besteht aus folgenden drei Teilen: der Angabe des Modellelementes, auf das er sich bezieht, einer Fehlermeldung oder Warnung, die im Falle des Nichteinhaltens ausgegeben wird, sowie der eigentlichen Bedingung. Zur Deklaration können, wie

[1]Fully Qualified Domain Name

schon in Xpand und Xtend, Elemente der Expressions-Subsprache verwendet werden, wodurch die Syntax der aus der Java-Modellierung bekannten Constraintsprache OCL sehr ähnlich ist. Ein einfaches Beispiel könnte so aussehen: In diesem Fall wird ein Cons-

```
context ModelElement ERROR
    "Name of element "+name+" ist too short." :
    name.length>1;
```

Abbildung 3.3: Beispiel für einen einfachen Constraint

traint für Elemente des Typs `ModelElement` deklariert. Die Einschränkung ist, dass das Attribut `name` des Elements eine Länge von mindestens zwei Zeichen haben muss. Wird die Bedingung nicht erfüllt, so wird die Bearbeitung abgebrochen und die entsprechende Fehlermeldung ausgegeben. Neben `ERROR` gibt es noch das Schlüsselwort `WARNING`, welches ebenfalls eine Fehlermeldung ausgibt, die Bearbeitung aber nicht unterbricht.

3.1.5 Workflowdateien als Verknüpfung der Sprachen

Für sich allein betrachtet sind Xpand, Xtend und Check lediglich drei deklarative Sprachen. Um nun das eigentliche Ziel, nämlich die Erzeugung von Quellcode zu erreichen, ist eine Laufzeitumgebung notwendig, die das erstellte Gerüst aus Templates, Extensions und Constraint-Dateien verknüpfen kann. Zu diesem Zweck werden Elemente der Eclipse MWE[2] verwendet. Die MWE beinhaltet neben den eigentlichen Codegeneratoren, die aus einem konkreten Modell und gegebenen Templates die letztendlichen Ausgabedateien erzeugen, viele weitere Treiberklassen beispielsweise zur Validierung von Modellen anhand von Constraint-Dateien, aber auch Hilfsmodule, wie zum Beispiel Code-Beautifier zur Verbesserung der Lesbarkeit von erzeugtem Quellcode. Der Ablaufplan, mit dem die einzelnen Module der Laufzeitumgebung zusammenarbeiten, wird in Form von workflow-Dateien festgelegt. Eine workflow-Datei ist ein XML-Dokument, in dem die benötigten Komponenten in der Reihenfolge des gewünschten Aufrufs instanziiert und konfiguriert werden. Ein einfaches Beispiel ist in Abbildung 3.4 dargestellt. Hier wird zunächst das Metamodell ausgewählt, das die Struktur des konkreten Modells vorgibt. Danach wird das erstellte Modell geladen, geparst und mit Hilfe eines entsprechenden Moduls und gegebener Check-Constraints validiert. Zuletzt wird ein Standard-Codegenerator instanziiert, der die Templates auf das geladene Modell anwendet und die Ausgabe im Ordner „src-gen“ ablegt. Statt der üblichen Endung `.xml` verwenden workflow-Dateien zur Abgrenzung die Endung `.mwe`. Solche Dateien sind dann in einer entsprechend konfigurierten Eclipse-Umgebung ausführbar.

[2]Modeling Workflow Engine

```
<?xml version="1.0"?>
<workflow>
    <!-- set up EMF for standalone execution -->
    <bean class="org.eclipse.emf.mwe.utils.StandaloneSetup" >
        <platformUri value=".."/>
    </bean>

    <!-- Standard-Metamodell auswählen -->
    <bean id="mm_emf" class="org.eclipse.xtend.typesystem.emf.EmfRegistryMetaModel"/>

    <!-- Konkretes Modell laden und in Slot "model" ablegen -->
    <component class="org.eclipse.emf.mwe.utils.Reader">
        <uri value="platform:/resource/MyModel.xmi" />
        <modelSlot value="model" />
    </component>

    <!-- check model -->
    <component class="org.eclipse.xtend.check.CheckComponent">
        <metaModel idRef="mm_emf"/>
        <checkFile value="metamodel::Checks" />
        <emfAllChildrenSlot value="model" />
    </component>

    <!--  generate code -->
    <component class="org.eclipse.xpand2.Generator">
        <metaModel idRef="mm_emf"/>
        <expand
            value="templates::root::main FOR model" />
        <outlet path="src-gen"/>
    </component>
</workflow>
```

Abbildung 3.4: Beispiel für eine workflow-Datei

3.2 Konfigurationsmittel in VHDL

Abgesehen von Werkzeugen zur Codegenerierung bietet VHDL auch von sich aus schon einige Mittel zur Parametrisierung und Konfiguration von Quellcode. Im Wesentlichen eignen sich dazu zwei Elemente der Sprache. Auf der einen Seite sind das die sogenannten *Generics*, andererseits das `generate`–Konstrukt. Bei Generics handelt es sich um einfache Parameter, die in der Entity-Deklaration einer Komponente deklariert und bei der Instanziierung festgelegt werden müssen. Innerhalb der Instanz können diese Parameter dann wie Konstanten verwendet werden. Abbildung 3.5 zeigt ein simples Beispiel. Die generate–Anweisung kann benutzt werden, wenn gleiche Codesegmente mehrfach

```
entity my_device is
    generic(
        my_parameter : integer range 0 to 127
    );
    port(
        clk : in std_logic
    );
end entity my_device;
```

```
architecture RTL of my_other_device is

    component my_device
        generic (my_parameter: integer range 0 to 127);
        port (clk: in std_logic);
    end component my_device;

begin
    my_instance : my_device                 -- instanziiere my_device
        generic map(my_parameter => 64)     -- setze Parameter auf 64
        port map(clk => clk);

end architecture RTL;
```

Abbildung 3.5: Ein einfaches Generics-Beispiel. Links die entity-Deklaration, rechts die Instanziierung.

erzeugt werden sollen. Beispielsweise wäre dies denkbar, um eine bestimmte Anzahl von D-Flipflops innerhalb eines FIFO-Speichers einzufügen. In diesem Fall würde sich die iterative Variante der Anweisung anbieten, die im Wesentlichen eine for-Schleife realisiert. Neben der iterativen Variante gibt es noch die konditionale generate–Anweisung. Sie entspricht grundsätzlich einer if–Abfrage. Diese beiden Anweisungstypen können insbesondere ineinander verschachtelt werden. Eine Beschreibung der Syntax ist in Abbildung 3.6 zu sehen. Verwendet man als Obergrenze des Intervalls des iterativen generate-

```
generate_label :
for identifier in discrete_range generate
    { concurrent_statements }
end generate [ generate_label ];

generate_label :
if boolean_expression generate
    { concurrent_statements }
end generate [ generate_label ];
```

Abbildung 3.6: Die Syntax des generate–Statements in VHDL

Statements eine Generic-Konstante, lassen sich auf einfache Weise bei der Instanziierung interne Strukturen, wie z.B. Registerbreiten festlegen.

4 Analyse

Dieses Kapitel beschreibt die Recherche und Domänenanalyse, die die Grundlage für den Entwurf der UART-Komponente darstellen. Da es eines der Hauptziele dieser Arbeit ist, eine hochkonfigurierbare Hardwarekomponente zu entwickeln, ist die Anforderungsanalyse gerade bezüglich der Konfigurationsoptionen ein wichtiger Teil des Entwicklungsprozesses. Um ein hohes Maß an Konfigurierbarkeit zu erreichen, ist es erforderlich eine möglichst große Menge von Anwendungsfällen abzudecken. Um dies zu erreichen, wird in diesem Kapitel zunächst der Funktionsumfang von UARTs aus Mikrocontrollern verschiedener Hersteller und Modellreihen betrachtet. Des Weiteren wird, zur Ergänzung der Ergebnisse dieser Analyse, eine Auswahl von Projekten betrachtet, die sich ebenfalls mit der Entwicklung von UARTs in Hardwarebeschreibungssprachen beschäftigen.

4.1 UARTs gängiger Mikrocontroller

Um festzustellen, welchen Funktionsumfang durchschnittliche Mikrocontroller-UARTs bieten, liegt es nahe, zunächst verbreitete industrielle Lösungen zu betrachten. Zu diesem Zweck wird im folgenden ein Überblick über verschiedene solcher Geräte vermittelt.

4.1.1 Atmel ATmega8/16/32

Bei der ersten betrachteten Familie von Mikrocontrollern handelt es sich um Varianten des ATmega Mikrocontrollers der Firma Atmel mit jeweils 8, 16 bzw. 32 KB Programmspeicher. Diese Modelle basieren auf der verbreiteten AVR-Architektur. Der USART dieser Modelle hat gemäß dem offiziellen Datenblatt [22] folgende Fähigkeiten:

- Synchrone und asynchrone Kommunikation
- Receiver/Transmitter einzeln aktivier– bzw. deaktivierbar
- konfigurierbare Baudrate
- Kommunikationsformat
  - 1 oder 2 Stoppbits
  - Framegröße 5, 6, 7, 8 oder 9 Datenbits
  - Paritätsmodus (odd/even)
- Fehlererkennung

  - Paritätskontrolle
  - Überlauferkennung
  - Erkennung von Formatfehlern (falsche Startbit– und Stoppbitpegel)
- Interrupts
  - TX Eingabe leer
  - Übertragung abgeschlossen
  - Empfang abgeschlossen
- Pufferung von empfangenen Daten durch 2-stufigen Ringpuffer
- Besondere Features
  - Modus für doppelte Geschwindigkeit (nur asynchron)
  - Tiefpassfilter/Rauschfilter zur Erkennung „falscher“ Startbits

4.1.2 NXP LPC2377/78

Die LPC2377 und LPC2378 Mikrocontroller der Firma NXP Semiconductors sind auf Anwendungen mit hohen Anforderungen an serielle Kommunikationsschnittstellen ausgelegt. Sie basieren auf einem ARM7TDMI-S Prozessor und sind mit einer großen Menge an Peripheriegeräten ausgestattet. Die Mikrocontroller besitzen vier intern baugleiche UARTs von denen einer (UART1) zusätzliche Hardware-Handshake-Leitungen zur Flusskontrolle bei der Kommunikation mit Modems besitzt. Ein weiterer UART (UART3) besitzt eine Infrarotschnittstelle zur Kommunikation über IrDA[1]. Im offiziellen Datenblatt [23] und im Benutzerhandbuch [24] ist folgender Funktionsumfang aufgestellt:

- Transmitter deaktivierbar
- 16C550-konforme Konfigurationsregister
- konfigurierbare Baudrate
- Kommunikationsformat
  - 1 oder 2 Stoppbits
  - Framegröße 5, 6, 7, 8 oder 9 Datenbits
  - Paritätsmodus (odd/even)
- Je 16 Byte FIFO-Puffer für Receiver und Transmitter
- konfigurierbare Trigger-Level für FIFOs bei 1, 4, 8 sowie 14 Byte

[1] Infrared Data Association

- Interrupts
  - Autobaud Zeitüberschreitung
  - Autobaud-Detektion abgeschlossen
  - Daten empfangen
  - Kombinierter Interrupt für Übertragungsfehler mit Statusregister
- Besondere Features
  - optionale Software-Flusskontrolle
  - automatische Baudratenerkennung (ABR, autobaud)

4.1.3 Microchip PIC32

Die PIC32-Serie der Firma Microchip Technology Inc. besitzt eine 32-bit Architektur basierend auf einem MIPS M4K Prozessorkern. Die Mikrocontroller verfügen über sechs UARTs mit jeweils folgendem Funktionsumfang (vgl. [25]):

- UART komplett aktivier–/deaktivierbar
- konfigurierbare Baudrate
- Kommunikationsformat
  - 1 oder 2 Stoppbits
  - Framegröße 5, 6, 7, 8 oder 9 Datenbits
  - Paritätsmodus (odd/even)
- Je 8-stufiger FIFO-Puffer für Receiver und Transmitter
- Interrupts
  - Daten empfangen
  - Trasmitter-Puffer leer
  - Paritätsfehler
  - Formatfehler
  - Pufferüberlauf
- Besondere Features
  - Autobaud–Unterstützung
  - Unterstützung für Hardwareflusskontrolle in jedem UART
  - IrDA
  - RX/TX-Interrupts einzeln aktivierbar

4.2 IP-Core UARTs

Wie bereits im ersten Kapitel thematisiert, wird die Entwicklung von Hardware mit Hilfe von Beschreibungssprachen immer relevanter. In Ergänzung zur Betrachtung der industriellen Hardwarelösungen soll dieser Abschnitt die Ansätze zweier verschiedener Projekte präsentieren, die sich mit dem Entwurf von UARTs als IP-Komponenten befassen.

4.2.1 UART16750

Das Projekt *UART16750* von Sebastian Witt [26] ist zum Zeitpunkt des Entstehens dieser Arbeit abgeschlossen und wurde zuletzt im April 2011 aktualisiert. Ziel des Projektes war es einen UART zu entwickeln, der der Spezifikation der Modelle 16550 bzw. 16750 der US-amerikanischen Firma National Semiconductor unterliegt. Die im Rahmen des Projekts entstandene VHDL-Komponente hat folgende Merkmale:

- Pins und Register kompatibel mit NS16550/16750
- konfigurierbare Baudrate
- reichhaltige Möglichkeiten der Hardwareflusskontrolle
  - RTS/CTS
  - DTR/DSR/RI/DCD
- Kommunikationsformat
  - 1, 1,5 oder 2 Stoppbits
  - Framegröße 5, 6, 7 oder 8 Datenbits
  - Paritätsmodus (odd/even)
- wahlweise keine, 16 oder 64 Byte FIFOs
- zahlreiche Trigger-Level für Receiver-FIFO
- Interrupts
  - Daten empfangen
  - Transmitterregister leer
  - Übertragungsfehler
- Besondere Features
  - synchrone Kommunikation
  - Verwaltung von Interruptprioritäten in Hardware (fest)

Das vollständige Projekt ist bei der *opencores.org*-Community zu finden, siehe [26].

4.2.2 UART 16550 core

Das Projekt *UART 16550 core* [27] von Jacob Gorban, Igor Mohor und Tadej Markovic beschäftigt sich mit der Entwicklung eines UART in Verilog. Es handelt sich hier um ein älteres Projekt, als beim UART16750. Das Projekt exisiert seit 2001 und ist zum Zeitpunkt dieser Arbeit im Entwicklungsstatus *stable*, wobei letzte Änderungen am Repository im Dezember 2010 stattfanden. Anhand der Statusberichte auf der Projektseite ist jedoch anzunehmen, dass der Quellcode im Jahr 2002 nahezu finalisiert wurde. Ähnlich, wie das Projekt von Sebastian Witt, ist es hier eines der Ziele kompatibel zur NS16550A–Spezifikation zu sein. Merkmale dieser IP-Komponente:

- Pins und Register (fast)[2] kompatibel mit NS16550
- RTS/CTS und DTR/DTS Kontrollleitungen
- konfigurierbare Baudrate
- Kommunikationsformat
  - 1, 1,5 oder 2 Stoppbits
  - Framegröße 5, 6, 7 oder 8 Datenbits
  - Paritätsmodus (odd/even)
- Je 16 Byte FIFO für Receiver und Transmitter, nicht deaktivierbar
- Trigger-Level für Receiver FIFO bei 1, 4, 8 und 14 Byte
- Interrupts
  - Alle Interrupts sind einzeln aktivierbar
  - Fünf Interrupt-Typen mit Detailinformationen in Statusregister
- Besondere Features
  - Interrupt-Prioritäten in Hardware (fest)
  - Unterstützt 8-Bit und 32-Bit Wishbone Bus
  - Zusätzliche Register mit Debuginformationen im 32-Bit-Modus

[2] Angabe der Projektbetreuer

4.3 Zusammenfassung und Merkmalmodell

Nach Betrachtung der verschiedenen UARTs zeigen sich zunächst viele Gemeinsamkeiten. Darunter fallen besonders die konfigurierbare Baudrate und die Formatierungsoptionen bezüglich der seriellen Kommunikation. Fast alle betrachteten UARTs unterstützen eine Datenwortbreite von 5 bis 8 oder 9 bit. Dazu kommen die Wahl von unterschiedlichen Paritätsmodi sowie die Option ein bzw. zwei oder 1,5 Stoppbits zu verwenden. Unterschiede gab es besonders im Bereich der Pufferung von eingehenden Daten, Verwaltung und Priorisierung von Interrupts sowie im Bereich der unterstützten Schnittstellen. Auch die Möglichkeit einzelne Komponenten auszuschalten bzw. Interrupts einzeln zu aktivieren war nicht immer gegeben. Insgesamt sind aus den Ergebnissen der Betrachtung die nun folgenden Merkmaldiagramme entstanden. Die Darstellung gliedert sich aus Platzgründen in ein Hauptdiagramm und zwei Subdiagramme. Die Subdiagramme beschäftigen sich dabei auf der einen Seite mit der Kommunikation und dem Format dieser, auf der anderen Seite mit der Behandlung von Interrupts.

Im Hauptdiagramm ist zunächst abzulesen, dass jeder UART eine parallele und serielle Schnittstelle besitzen muss. Da es die Hauptaufgabe eines UART ist, serielle und parallele Daten in beide Richtungen zu übersetzen, wäre das Modul andernfalls auf einer Seite abgetrennt und könnte seine Arbeit nicht verrichten. Insgesamt gibt es verschiedene Möglichkeiten für parallele respektive serielle Schnittstellen, diese sind allerdings im Diagramm als Alternativen aufgetragen, um deutlich zu machen, dass zu einer Zeit nur eine aktiv sein kann. Weiterhin muss jeder UART mindestens eine Receiver- oder Transmittereinheit oder beides besitzen. Es sind Fälle denkbar, in denen ein UART nur für den Empfang oder nur zum Senden von Daten benötigt wird, weshalb nicht beide notwendig sind. Keines der beiden Elemente zu haben wäre aber aus naheliegenden Gründen nicht sinnvoll. Sowohl die Empfängereinheit, als auch der Sender können mit FIFO-Puffern ausgestattet sein, die eine feste Größe haben. Im Falle des Empfängers ist es außerdem nötig einen Trigger-Level anzugeben, bei dem ein Interrupt ausgelöst wird, um zu signalisieren, dass Daten zur Abholung bereit sind. Dies gilt selbstverständlich nur für den Fall, dass das Gerät Hardwareunterbrechungen unterstützt und diese auch aktiviert sind.

Da jeder UART eine Form von Kommunikation ausführt, sei es Senden oder Empfangen, ist es obligatorisch, dass die Parameter dieser Kommunikation festgelegt sind. Die Details der Kommunikation sind daher im folgenden Subdiagramm dargestellt. Ein UART kann zunächst einen synchronen Kommunikationsmodus bieten. Dieser ist als optional markiert, da nicht alle betrachteten Geräte diese Fähigkeit haben. Eine weitere optionale Fähigkeit besteht im Anbieten einer Schnittstelle für Hardwareflusskontrolle. Von den meisten Geräten wird eine einfache Flusskontrolle über die Signale RTS (ready-to-send) und CTS (clear-to-send) angeboten. Besonders für den Betrieb in Kombination mit einem Modem wäre allerdings eine weitreichendere Flusskontrolle angebracht. Jeder UART muss eine bestimmte Symbolrate unterstützen. Bei allen betrachteten, besonders bei NS16550-kompatiblen Geräten, war die Baudrate im Betrieb über ein Register konfigurierbar, in das ein Teilungsfaktor für den Systemtaktgeber eingetragen werden muss.

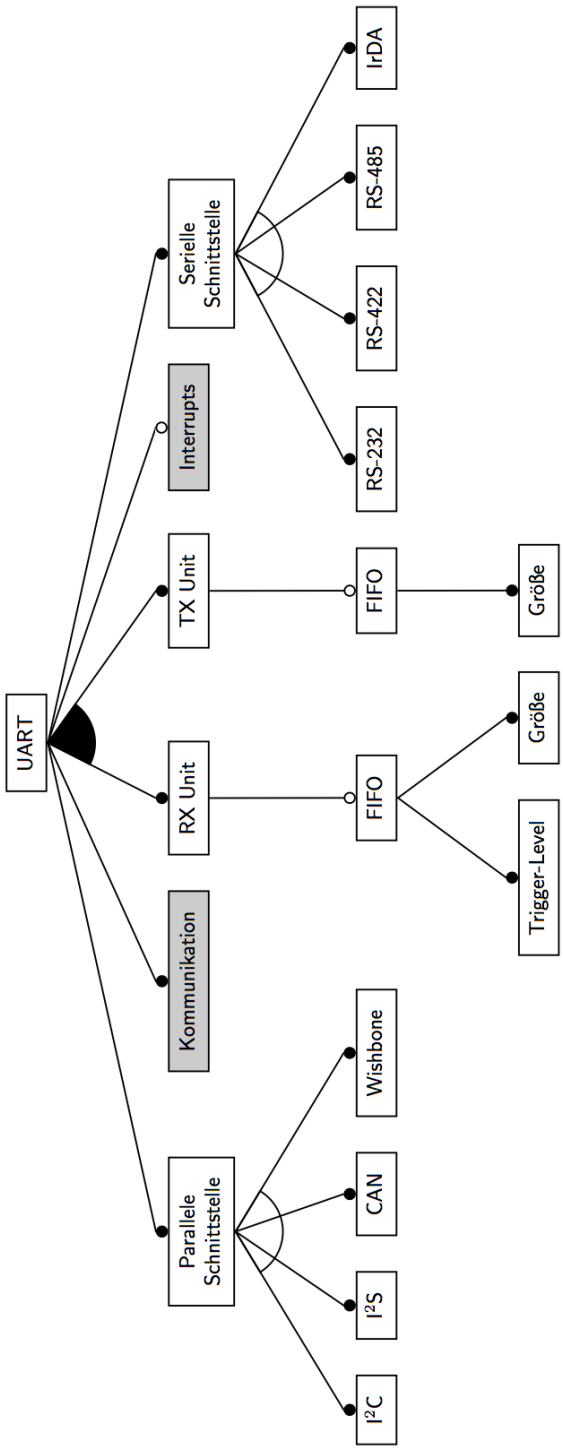

Abbildung 4.1: Merkmalmodell für einen UART

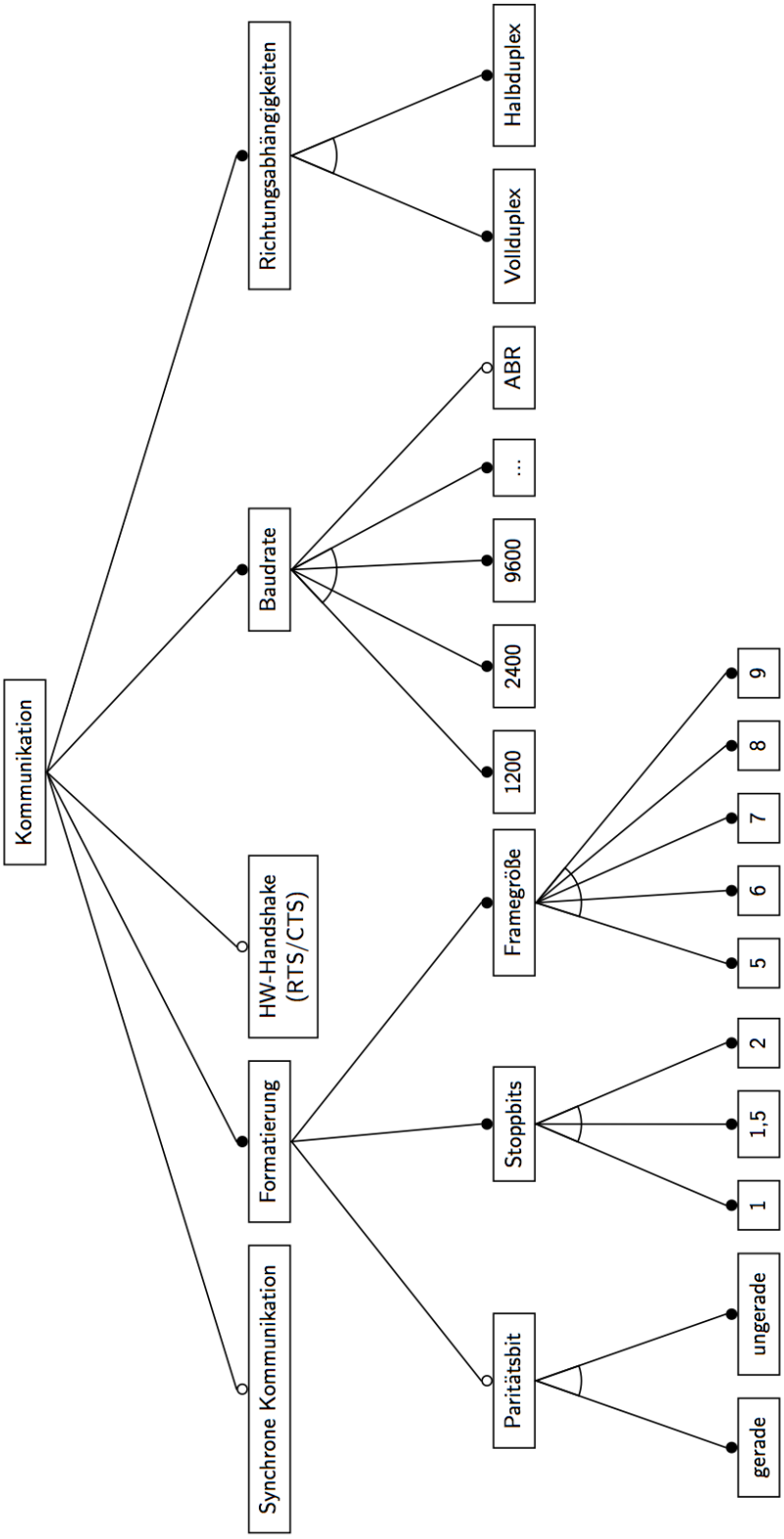

Abbildung 4.2: Subdiagramm für kommunikationsbezogene Merkmale

Einige der Geräte unterstützten darüberhinaus auch die automatische Erkennung der Symbolrate anhand eines eingehenden Zeichenstroms (Autobaud oder ABR). Bezüglich der Richtungsabhängigkeiten ist bei neueren UARTs fast nur noch ein Vollduplexbetrieb, also gleichzeitiges Senden und Empfangen von Daten vorgesehen. Gerade ältere Geräte konnten aber oft auch nur im Halbduplex-Modus betrieben werden. Die Formatierungsmerkmale eines UART stellen sich wie folgt dar. Die Breite des Datenwortes und damit die Framegröße liegen bei allen betrachteten Geräten zwischen 5 und 9 bit. Das neunte Datenbit dient dabei nur zur Abgrenzung von Daten- und Adressinformation bei Konfigurationen mit Multiprozessorkommunikation. Es gibt weiterhin die Möglichkeit ein Paritätsbit zur Validierung der übertragenen Daten mitzusenden, welches entweder als gerade oder ungerade Parität bezüglich der Anzahl der Einsen im Datenwort berechnet werden kann. Die Anzahl der Stoppbits ist variabel zwischen 1, 1,5 und 2. Die Angabe von 1,5 Stoppbits meint dabei, dass der Pegel des Signals 1,5-mal so lange gehalten wird, wie bei einem Stoppbit.

Die Kommunikation mit dem UART von Seiten der parallelen Schnittstelle, insbesondere die Abholung empfangener Daten, kann entweder im Polling-Betrieb, also mit wiederholter Abfrage, oder über Hardwareunterbrechungen realisiert werden. Die Merkmale eines UARTs mit Interrupt-Unterstützung sind im entsprechenden Subdiagramm in Abbildung 4.3 dargestellt. Die Verwaltung von Interrupts variiert stark von Gerät

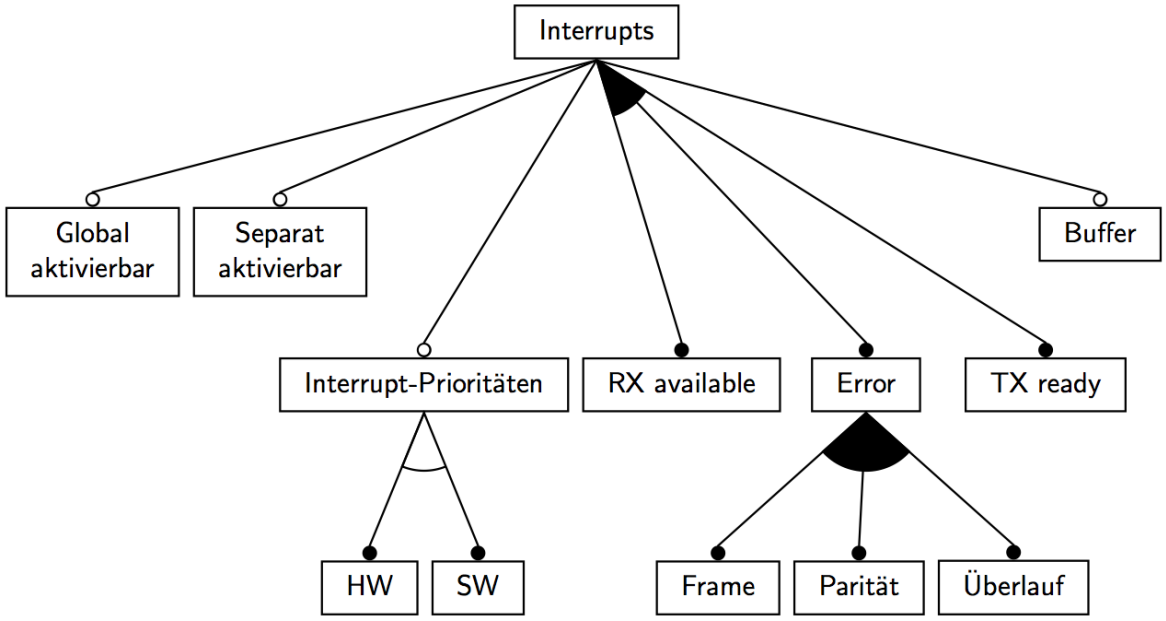

Abbildung 4.3: Subdiagramm für Interrupt-bezogene Mermale

zu Gerät. Welche Arten von Interrupts ein Gerät unterstützt ist sehr verschieden. Man kann sie jedoch in drei wesentliche Gruppen zusammenfassen. Zum einen gibt es Interrupts, die den Empfang von Daten bzw. den Füllstand des Empfängerpuffers anzeigen, des Weiteren solche, die sich mit dem Füllstand und Status des Senders beschäftigen

und zuletzt eine Reihe von Fehlerzuständen. Letztere lassen sich dabei in die Gruppen Frame- und Formatfehler, Paritätsfehler und Pufferüberläufe einordnen. Letztendlich ist jede Kombination von Interrupts möglich, solange die entsprechenden Voraussetzungen im UART gegeben sind. Ein Interrupt für Pufferüberläufe wäre Beispielsweise ohne einen entsprechenden Puffer nicht sinnvoll. Im Allgemeinen gibt es aber die Möglichkeit Interrupts, wenn sie vorgesehen sind, global einzuschalten. Oft ist es außerdem möglich, über das Setzen von einzelnen Bits in einem Konfigurationsregister einzelne Interrupts zu aktivieren oder zu deaktivieren. Es ist weiterhin möglich Interrupts bestimmte Prioritäten zuzuordnen, diese können entweder direkt in Hardware realisiert sein oder in Software durch den Gerätetreiber geregelt werden. Bei Realisierung in Hardware sind die Prioritäten meist festgelegt und können nicht vom Benutzer verändert werden. Die Prioritätssteuerung in Software ist an dieser Stelle flexibler. Manche Geräte unterstützen auch die Pufferung von aufgetretenen Interrupts. Dies kann zum einen über ein Register erfolgen, in dem zumindest ein Interrupt jeder Art gespeichert wird oder auch in einem FIFO-Puffer.

5 Entwurf des UART

Dieses Kapitel beschreibt den Entwurf der UART-Komponente. Ausgehend von der im vorigen Kapitel dargestellten Domänenanalyse wird zunächst der Umfang der umzusetzenden Funktionen hergeleitet und erläutert, wie die Konfiguration der einzelnen Komponenten im Betrieb bzw. bei der Instanziierung stattfinden soll. Auf der Grundlage dieses Konfigurationsschemas wird dann das Metamodell für die Generierung des UART dargestellt. Im Folgenden wird die Ausarbeitung des Hardwaremodells, eines um Konfigurationspunkte erweiterten Blockdiagramms, erläutert, welches die Basis für die Umsetzung des UART in VHDL bietet.

5.1 Funktionsumfang des UART

Um einen zufriedenstellenden Konfigurationsumfang zu erreichen, liegt es nahe, zunächst die Merkmale, die alle im Laufe der Domänenanalyse betrachteten UARTs gemeinsam haben, als obligatorisch zu betrachten und mit in den Entwurf einzubeziehen. Ausgehend vom bisherigen Funktionsumfang des LavA-UART (vgl. Kapitel 2) gehören dazu zunächst:

- konfigurierbare Baudrate
- verschiedene Interrupts
- konfigurierbares Datenformat
  - variable Anzahl Stoppbits
  - Parität
  - variable Framegröße
- Pufferung von Daten (Senden und Empfang)

Zusätzlich zu diesen Features gilt es noch die jeweiligen Schnittstellen zu bestimmen. Auf der Systemseite soll hier als parallele Schnittstelle zunächst nur der bereits verwendete Wishbone-Bus weiter unterstützt werden. Eine Implementierung weiterer Busse wie CAN oder I^2C ist nicht vorgesehen. Bezüglich der seriellen Schnittstelle soll auch weitestgehend der bisherige Funktionsumfang erhalten bleiben. Diese Entscheidung liegt in der Tatsache begründet, dass es in erster Linie Ziel dieser Arbeit ist, den Funktionsumfang des LavA-UART durch Erweiterung um Konfigurationsmöglichkeiten innerhalb der Plattform flexibler zu machen und nicht eine möglichst hohe Kompatibilität

mit möglichst vielen verschiedenen Geräten zu erreichen. Neben der Datenübermittlung wird es allerdings die Möglichkeit der einfachen Hardwareflusskontrolle über die Signale RTS bzw. CTS geben, um Pufferüberläufen vorzubeugen. Weitere vereinzelt aufgetretene Features, wie automatische Baudratendetektion oder spezielle Debug-Register sind ebenfalls nicht geplant. Die Implementation eines Halbduplexmodus ist darüberhinaus nicht vorgesehen, da der Halbduplexbetrieb den effektiven Datendurchsatz halbiert, aber im Gegenzug keine nennenswerten Ressourcenvorteile bringt.

5.1.1 Formatierung und Kommunikation

Im Bereich der Formatierung der seriellen Aus- und Eingabe sollen fast alle in der Domänenanalyse aufgetretenen Möglichkeiten implementiert werden. Die Breite des Datenwortes soll zwischen fünf und neun Bit variabel sein. Hinzu kommen eine Stoppbit-Anzahl von ein oder zwei Bit und die Möglichkeit ein Paritätsbit, entweder mit gerader oder ungerader Parität, zu verwenden.

5.1.2 Interrupts

In seinem bisherigen Zustand bietet der LavA-UART eine Interrupt-Leitung in Verbindung mit zwei Zustandsbits innerhalb der Datenregister der Transmitter- und Empfängereinheiten. Über diese Architektur werden zwei verschiedene Interrupts realisiert, die die Ereignisse „Senden abgeschlossen“ und „Datenwort empfangen“ widerspiegeln. Durch die Erweiterung um Pufferspeicher und Maßnahmen der Fehlererkennung ist diese Modellierung nicht mehr ausreichend. Neben den bereits vorhandenen Interrupts sollen vier neue Fehlerzustände, nämlich Paritätsfehler im aktuellen Datenwort, ein Fehler im Format des Datenwortes (beispielsweise ein „falsches Startbit“) und Überlauf des Empfänger- bzw. Sendepuffers, hinzugefügt werden. Zwar wird weiterhin eine einzelne Interrupt-Leitung genutzt, diese wird aber mit einem dedizierten Statusregister verbunden, dass mit einem Bit pro Interrupt die aufgetretenen Unterbrechungen speichert. Die Interrupts sollen sowohl global, als auch separat aktiviert und deaktiviert werden können. Weiterhin soll es möglich sein diesen Interrupts verschiedene Prioritäten zuzuordnen. Die Verwaltung der Prioritäten und entsprechende Behandlung der Interrupts wird hierbei der Treibersoftware überlassen, um möglichst einfach eine benutzerdefinierte Priorisierung zu ermöglichen.

5.1.3 Pufferung

Bezüglich der Pufferung von Daten soll der UART für die Sender- und Empfängereinheit FIFO-Puffer mit variabler Größe unterstützen. Eine explizite Pufferung von aufgetretenen Interrupts ist nicht vorgesehen, wird aber teilweise durch den Einsatz des Interrupt-Statusregisters realisiert, da zumindest ein Interrupt jeder Art gleichzeitig gespeichert bleibt.

5.2 Umsetzung der Konfigurationspunkte

Nachdem der Funktionsumfang festgelegt ist, bleibt noch die Entscheidung, welche der Konfigurationsmöglichkeiten im Betrieb und welche nur zur Instanziierungszeit bestimmtbar sein sollen. Grundsätzlich ergibt es Sinn, Konfigurationspunkte, die direkten Einfluss auf die Ressourcenbelegung auf dem FPGA haben, zur Instanziierungszeit festzulegen. Komponenten, über die Ressourcen gespart werden können, wenn sie nicht benötigt werden sind auf oberster Ebene die Transmitter- und Receiver-Einheiten. Wird eine der beiden Funktionalitäten nicht benötigt, so können große Teile der Architektur von vornherein ausgelassen werden. Dies ist ebenfalls gegeben, wenn keine Interrupts gewünscht werden oder ein Paritätsbit nie verwendet werden soll. In diesen Fällen können ebenfalls Register und Module, wie der Paritätsgenerator, nicht erzeugt werden, um eine Ressourcenersparnis zu erreichen. Bei Interrupts und Paritäten ergibt sich die Besonderheit, dass neben der Konfiguration zur Modellierungszeit auch Einstellungen, wie das Aktivieren einzelner Interrupts oder das Ändern des Paritätsmodus, im Betrieb vorgenommen werden können. Diese Komponenten müssen also zu beiden Zeitpunkten manipuliert werden können. Parameter des UART, die zur Modellierungszeit festgelegt werden müssen, sind beispielsweise die Größe der FIFO-Puffer, da je nach Größe andere Hardwarestrukturen synthetisiert werden müssen. Der Trigger-Level ist wiederum eine Option die zur Laufzeit veränderbar sein kann. Insgesamt ergibt sich damit die Aufteilung, die in Tabelle 5.1 dargestellt ist.

Modellierung	**Betrieb**
Vorhandensein der Paritätsmodule (Generator und Überprüfungseinheit)	Paritätsmodus Aktivieren des Paritätsbits Framegröße Stoppbits Baudrate Hardwareflusskontrolle (RTS/CTS)
Vorhandensein der RX/TX-Einheiten FIFO-Puffer ja/nein Größe der FIFOs	RX-FIFO Trigger-Level
Interrupt-Unterstützung	Aktivierung von Interrupts Setzen der Prioritäten (in SW über Treiber)

Tabelle 5.1: Zuordnung der Konfigurationspunkte

5.3 Eclipse-Metamodell

Anhand der im vorigen Abschnitt dargestellten Aufteilung der Konfiguration in Merkmale, die im Betrieb manipuliert werden können und solche, die nur zum Zeitpunkt der

Zusammenstellung bzw. Initialisierung einer UART-Instanz bestimmt werden können, ist es nun bereits möglich, das Ecore-Metamodell zu bestimmen, das die Grundlage des Codegenerierungsprozesses bietet. Das Metamodell muss alle Konfigurierungsmöglichkeiten abdecken, die gemäß Tabelle 5.1 der Modellierung zugeordnet sind. Das Klassendiagramm des Metamodells ist in Abbildung 5.1 dargestellt.

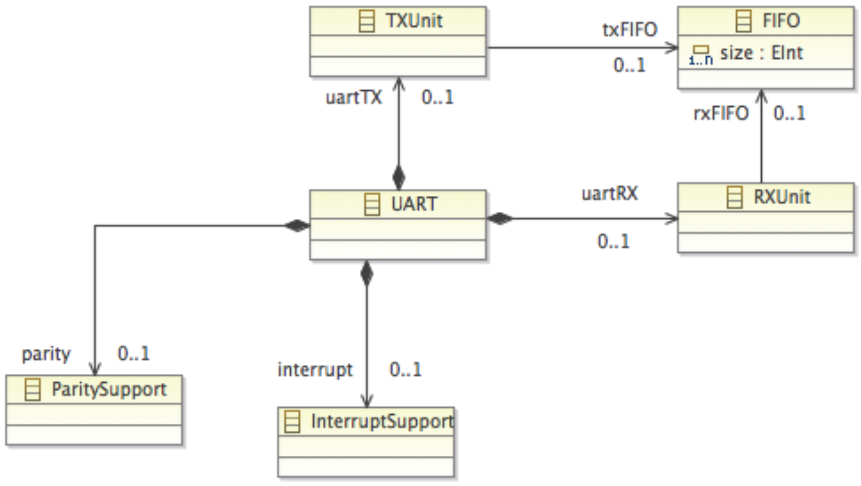

Abbildung 5.1: Klassendiagramm des Ecore-Metamodells

Demnach besteht ein UART aus einer RX-Einheit, einer TX-Einhei, Komponenten für die Verwaltung von Interrupts sowie zur Generierung und Überprüfung von Paritätsbits. Da alle diese Komponenten mit einer Multiplizität von 0..1 gekennzeichnet und somit optional sind, wäre ein konkretes Modell, das keine der Komponenten enthält zunächst valide. Daher ist es wichtig, einen entsprechenden Constraint mittels Check (siehe Abschnitt 3.1.4) zu formulieren, der sicherstellt, dass gemäß der Intention der Modellierung zumindest eine RX- oder TX-Einheit vorhanden sind.

5.4 Entwurf einzelner Komponenten

Aus den bisherigen Überlegungen lässt sich bereits eine relativ kanonische, grobgranulare Architektur des UART abzeichnen. Die zwei Hauptkomponenten des alten LavA-UART, nämlich Sende- und Empfangseinheit, können als zwei Hauptkomponenten des neuen UART übernommen werden. Sie müssen lediglich intern um die neuen Funktionalitäten erweitert werden. Durch die Einbringung der neuen Funktionen kommen viele Konfigurations- und Statusregister und neue Leitungen hinzu. Auch die Interaktion des UART mit der parallelen Schnittstelle, darunter insbesondere das Lesen und Schreiben von Daten muss beachtet werden. Da diese Komponenten im Grunde das Verhalten des UART steuern, können sie aggregiert als Steuereinheit betrachtet werden. Der UART

lässt sich so überschneidungsfrei in die drei Komponenten RX-Einheit, TX-Einheit und Steuereinheit einteilen.

5.4.1 RX- und TX-Einheit

Die RX- und TX-Einheiten beinhalten alle Teile des UART, die sich mit dem Empfang bzw. im Falle der TX-Einheit mit dem Senden serieller Daten beschäftigen. Im Wesentlichen gehören dazu die Serien-Parallel- bzw. Parallel-Serien-Wandler, die das Übersetzen von seriellen auf parallele Daten übernehmen sowie die neu hinzugekommenen FIFO-Speicher. Beide Einheiten sind über Steuereingänge mit der Steuereinheit verbunden, um für das Senden und Empfangen wichtige Formatkonfigurationsdaten zu erhalten. Ebenfalls Teil der RX-Einheit ist ein Paritätsprüfer, der zur Validierung von empfangenen Daten dient. Entsprechend hierzu enthält die TX-Einheit einen Paritätsgenerator. Beide Einheiten werden des Weiteren um neue Statusleitungen ergänzt. Im Falle der RX-Einheit muss beispielsweise die Information über Korrektheit des aktuellen Datenwortes an die Steuereinheit weitergeleitet werden, damit diese im negativen Fall eine entsprechende Unterbrechung auslösen kann.

5.4.2 Steuereinheit

Den komplexeren Teil der Architektur stellt die Steuereinheit dar, da in ihr ein Großteil der Verhaltenslogik des UART liegt. Die Steuereinheit steuert zunächst das Verhalten der RX- und TX-Einheit. Zu diesem Zweck verfügt sie über ein Konfigurationsregister, in dem die Formatierungsoptionen des seriellen Datenstroms eingetragen sind. Die Bedeutung der einzelnen Bitpositionen ist dabei in Tabelle 5.2 dargestellt. Die einzelnen Einstellungen werden über entsprechende Leitungen an die Empfangs- und Sendeeinheit weitergeleitet.

Bitposition	**Belegung**
0	Handshake aktivieren
1	Stoppbits: ‘0’ = 1 Bit, ‘1’ = 2 Bit
2	Paritätsmodus: ‘0’ = gerade, ‘1’ = ungerade
3	Paritätsbit aktivieren
4-6	Framegröße
7-14	RX-FIFO Trigger-Level

Tabelle 5.2: Mapping des Format-Konfigurationsregisters

Neben der Formatierung verwaltet die Steuereinheit auch ein Baudratenregister, das den Taktteilungsfaktor für den Systemtaktgeber beinhaltet. Zwei Taktteiler sorgen für die Berechnung der RX- und TX-Taktraten, die ebenfalls an die jeweiligen Einheiten weitergegeben werden. Die Steuerung der Unterbrechungen geschieht mit Hilfe zweier weiterer Register. Auf der einen Seite verwaltet die Steuereinheit ein Interrupt-Konfigurationsregister, durch welches einzelne Interrupts aktiviert bzw. deaktiviert wer-

den können. Ein Interrupt-Statusregister verwaltet die aufgetretenen Interrupts. Ist aus den Ausgaben der Kommunikationseinheiten eine Situation abzulesen, die eine Unterbrechung erfordert, so wird das entsprechende Bit im Statusregister gesetzt. Beide Register verwenden dabei die gleichen Bitzuordnungen, die in Tabelle 5.3 aufgeführt sind.

Bitposition	**Interrupt**
0	Überlauf des RX-FIFO
1	Überlauf des TX-FIFO
2	Formatfehler
3	Paritätsüberprüfung fehlgeschlagen
4	TX-FIFO leer (keine Daten zu übertragen)
5	RX-FIFO Füllstand über Trigger-Level oder voll

Tabelle 5.3: Mapping der Interrupt-Register

5.4.3 Anbindung an den Wishbone-Bus

Die Spezifikation des Wishbone-Busses [16] bietet verschiendene Architekturen zur Verbindung von kommunizierenden Geräten. Es wird dabei grundsätzlich aber nur zwischen Master- und Slave-Geräten unterschieden. Der UART nimmt grundsätzlich die Rolle eines Slave ein, was die Anbindung an den Datenbus unkompliziert macht, da die Initiierung einer Kommunikationsphase nie vom UART selbst ausgeht. Der UART muss lediglich die eingehenden Wishbone-Steuersignale überwachen und bei Beginn eines Kommunikationszyklus abhängig von der gegebenen Adresse und dem Pegel des WE[1]-Signals eine entsprechende Aktion ausführen. Die Aktionen die über den Bus ablaufen sind dabei:

- Lesen von empfangenen Daten
- Auslesen des Interruptsstatus
- Schreiben von zu übertragenden Daten
- Schreiben und ggfs. Lesen der Konfigurationsregister

Dementsprechend verwendet der UART die Adresszuordnung, die in Tabelle 5.4 dargestellt ist. Das erste Zeichen der Adresse, hier immer die 8, gibt an, dass es sich um einen Wishbone-Zugriff handelt. Die folgenden zwei Zeichen der Adresse sind Platzhalter für den Gerätetyp, wobei ein UART durch die Bitfolge „01“ kodiert wird. Das vierte und fünfte Zeichen (X) muss durch die jeweilige Instanz des UART im MPSoC, beginnend bei 1, ersetzt werden. Eine vollständige Adressierung für das Baudratenregister der zweiten UART-Instanz eines Gerätes würde dementsprechend „0x80102000“ lauten.

[1] Write Enable

Adresse	Bedeutung
0x801XX000	Baudratenregister
0x801XX004	RX-Daten
0x801XX008	TX-Daten
0x801XX00C	Formatierungsregister
0x801XX010	Interruptkontrollregister
0x801XX014	Interruptsstatusregister

Tabelle 5.4: Mapping der Speicheradressen des UART

5.4.4 Blockdiagramm des UART

Abbildung 5.2 zeigt das Blockdiagramm des UART gemäß des Entwurfs der einzelnen Komponenten. Optionale Teile der Architektur sind dabei mit Ausnahme der kompletten RX- und TX-Einheiten in grün dargestellt. Die Auslassung zwischen den einzelnen FIFO-Slots spiegelt die variable Größe wieder. Aus Gründen der Übersichtlichkeit sind besonders einzelne Leitungen, wie zum Beispiel die Steuereingänge der Multiplexer nicht mit eingezeichnet.

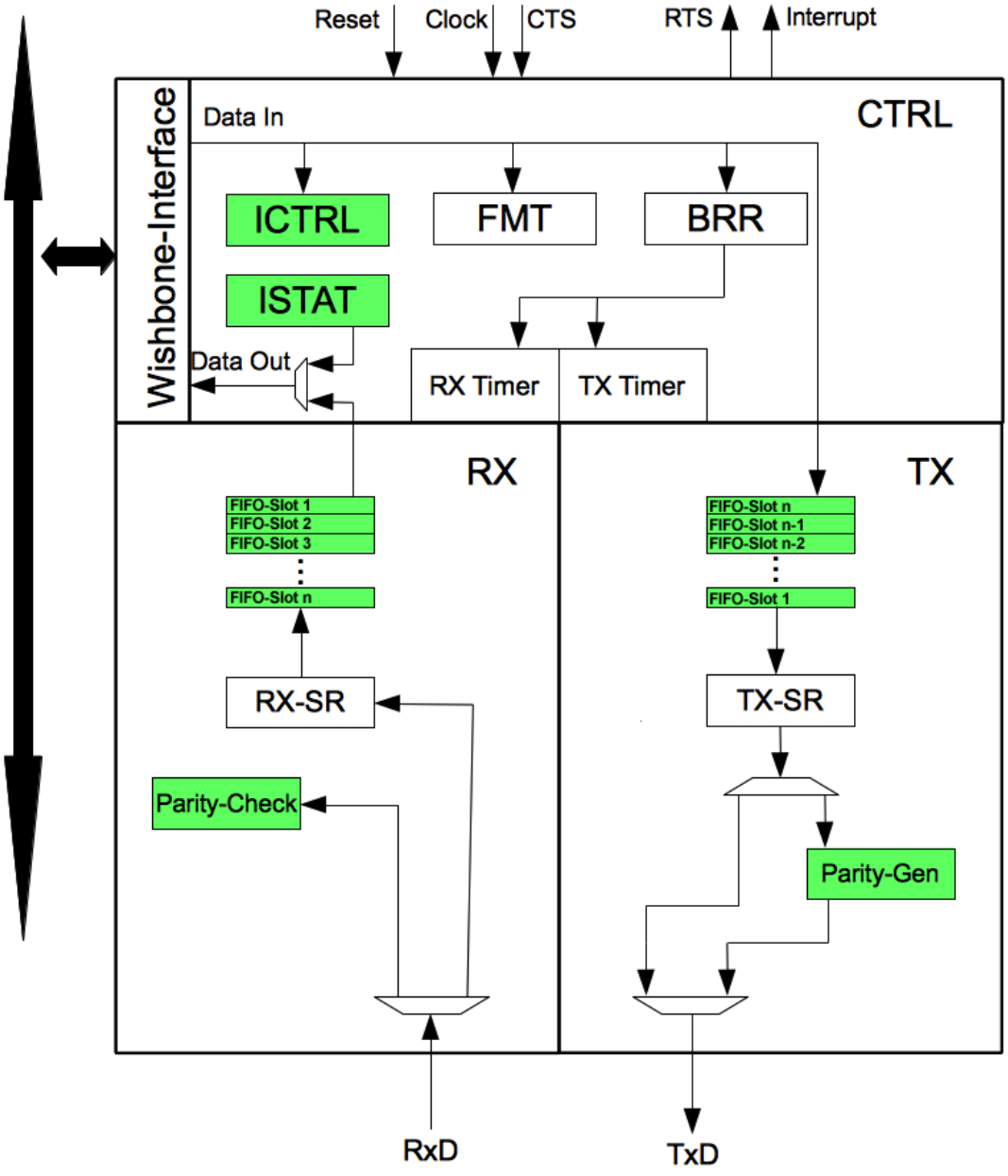

Abbildung 5.2: Blockdiagramm des UART

6 Implementation

Ausgehend von den Überlegungen und Modellierungen aus dem vorigen Kapitel beschreibt dieses Kapitel die Entwicklung des UART und der zur Codeerzeugung benötigten Dateien. Zu diesem Zweck wird zunächst die Umsetzung der komplexeren Teile der Implementation in VHDL betrachtet. Im folgenden wird erläutert, wie die entsprechenden Code-Artefakte durch Einsatz der Eclipse Modeling Tools erzeugt werden. Zum Abschluss des Kapitels wird schließlich die Treibersoftware zur Ansteuerung des UART dargestellt.

6.1 Implementation einzelner Komponenten in VHDL

Die Konzeption einer Hardwareproduktlinie erfordert die Entwicklung modularer Komponenten, aus denen sich ein anwendungsspezifisches Produkt maßschneidern lässt. Um ein gutes Bild zu vermitteln, wie dies im Falle der UART-Familie geschehen ist, soll an dieser Stelle die Implementation der wesentlichen Entitäten vorgestellt werden. Zur Verbesserung der Verständlichkeit der Interaktion der einzelnen Komponenten wird dabei im Zweifelsfall davon ausgegangen, dass ein UART mit allen möglichen optionalen Komponenten vorliegt, anstatt immer eine Fallunterscheidung zu machen.

6.1.1 Die Top-Level-Entität

Die Top-Level-Entität stellt den eigentlichen UART dar und bietet nach außen die parallele und serielle Schnittstelle an. Da der UART auf der parallelen Seite das Wishbone-Protokoll implementiert, verfügt die UART-Entität über die hierfür erforderlichen Ein- und Ausgabesignale. Dazu gehören Adress- und Datenleitungen und die Steuersignale *WE* (write enable), *STB* und *CYC* auf der Eingabeseite sowie Datenleitungen und *ACK*-Signal auf der Ausgabeseite (vgl. [16]). Bezüglich der seriellen Kommunikation gibt es zwei Leitungen zum Senden und Empfangen sowie zwei Leitungen für Hardwareflusskontrolle (*RTS/CTS*). Strukturell beinhaltet die Entität drei Hauptkomponenten: Steuereinheit, RX-Einheit und TX-Einheit. Funktional hat sie neben der Verknüpfung der Untereinheiten die Aufgabe, eine Übersetzung des Wishbone-Protokolls auf interne Steuersignale durchzuführen und umgekehrt, die dann an die Untereinheiten weitergegeben werden können. Wie in Kapitel 5 dargestellt, gibt es maximal sechs einzeln adressierbare Register im UART. Das sind zunächst die RX- und TX-Datenregister sowie die drei Konfigurationsregister für Baudrate, Kommunikationsformat, Interrupts und das Interruptstatusregister. Da die Wishbone-Ansteuerung des UART vom MPSoC schon in sofern gefiltert wird, dass nur für den UART relevante Anweisungen in Verbindung

mit einem *CYC*-Signal auftreten und eine Busbreite von 32 Bit vorliegt, reicht es an dieser Stelle einfach die Adressbits an den Positionen 4 bis 2 zu überprüfen, um das gewünschte Register zu erhalten. In Verbindung mit dem Pegel der *WE*-Leitung lässt sich so die gewünschte UART-Anweisung dekodieren und ein entsprechendes internes Signal aktivieren, welches die Abarbeitung der Anweisung in einer der Untereinheiten des UART anstößt. Während bei einem Schreibzugriff nichts weiter zu tun ist, wird im Falle eines Lesezugriffs noch das richtige Datum an den ausgehenden Datenleitungen angelegt.

6.1.2 RX-Einheit

Die RX-Einheit stellt den seriellen Empfänger des UART dar. Als Eingabesignale bekommt die Komponente einen Systemtakt und den reduzierten Baudratentakt, *reset* und *read* als Steuersignale, die serielle Datenleitung und die von der Steuereinheit ausgegebenen Konfigurationsparameter für Framegröße, Paritätsverhalten und Stopbitlänge. Die Ausgabe der RX-Einheit besteht aus Statusinformationen zu Pufferfüllstand und Auftreten eines Pufferüberlaufs sowie der parallelen Datenausgabe. Strukturell besteht die Einheit aus einem Schieberegister und einem Datenpuffer zum Empfangen der seriellen Eingabe sowie gegebenenfalls einem Paritätsprüfer.

6.1.2.1 Empfang von Daten

Die Verarbeitung von seriellen Eingaben wird innerhalb eines einzigen Prozesses erledigt. Zunächst werden die Konfigurationsparameter, die bisher nur als logische Werte vorliegen, in Variablen geschrieben. Dies geschieht zu jedem Taktzyklus, da sich diese Parameter zur Laufzeit jederzeit ändern können. Nach diesem Initialisierungsschritt beginnt die eigentliche Verarbeitung der Daten nach dem Schema für asynchronen seriellen Kommunikation des RS-232-Standards (vgl. Abbildung 6.1). Die aktuelle Bitposition im empfangenen Datenwort wird dabei in einer Variable gespeichert. Da das Verhalten der Komponenten in jedem Schritt von der Bitposition abhängt, realisiert dies eine einfache Zustandsmaschine. Solange der Zähler auf Position 0 steht und kein Startbitpegel auftritt, bleibt die RX-Einheit im Zustand *idle* und wartet auf Eingaben. Tritt ein Startbit auf, so beginnt ein neuer Verarbeitungszyklus und die Bitposition wird auf 1 gesetzt. Je nach eingestellter Wortbreite werden dann bis zum letzten Datenbit die Werte an die entsprechende Stelle eines Pufferregisters, das die Aufgabe des RX-Schieberegisters übernimmt, geschrieben. Ist die Bitposition bis zum Ende des Datenbitbereichs fortgeschritten, dann wird das empfangene Datenwort aus dem „Schieberegister“ in das Ausgaberegister übertragen. Wenn der UART für die Verwendung von Paritätsbits konfiguriert ist, wird das *enable*-Signal des Paritätsprüfers aktiviert, da das nächste Zeichen ein Paritätsbit sein muss und die Prüfung mit den nun kompletten Daten durchgeführt werden kann. Je nach Einstellung werden danach ein oder zwei Stoppbits erwartet. Trifft dies nicht zu, so wird an dieser Stelle der Interrupt für einen Framefehler erzeugt. Nach Abschluss der Verarbeitung wechselt die Bitposition wieder auf 0, also in den Zustand *idle*. Ist ein Interrupt für Paritätsfehler eingeschaltet, wird dieser nach Abschluss des

Empfangs erzeugt und bleibt aktiv bis ein neues Zeichen gelesen wurde.

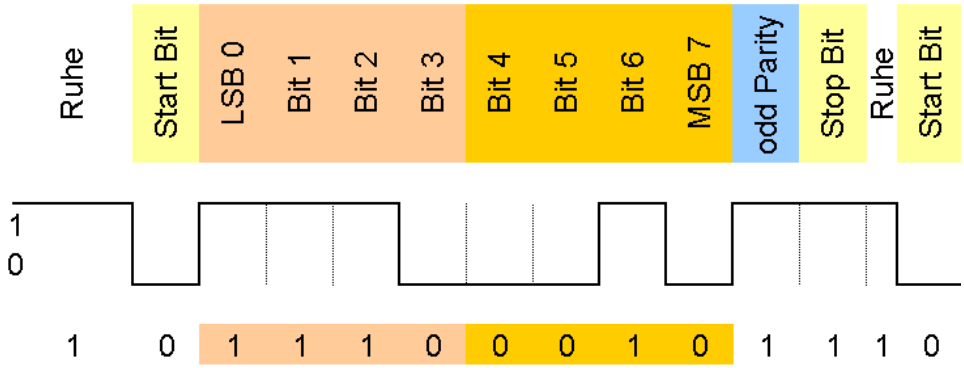

Abbildung 6.1: Timing-Schema nach RS-232. Quelle: [28]

6.1.2.2 Datenpufferung

Die Realisierung der Datenpufferung hängt von der Verwendung eines FIFO-Puffers ab. Wird kein Pufferspeicher verwendet, so bleibt das zuletzt empfangene Datenwort im Ausgaberegister bis es entweder vom MPSoC gelesen oder durch den Empfang eines weiteren Datums überschrieben wird. Bei Verwendung von Pufferspeicher sorgt der Empfang eines Datenwortes einen Peak auf der *write*-Leitung des FIFO und das neue Datum wird im Puffer gespeichert. Ist der Puffer voll wird zyklisch das älteste Zeichen überschrieben. Der Interrupt *RXREADY*, der darauf hinweist, dass Daten abgeholt werden können, wird je nach Einstellung nach dem Empfang einer gewissen Anzahl von Zeichen ausgelöst. Bei Verwendung von Pufferspeicher handelt es sich hier um den eingestellten Trigger-Level, andernfalls wird der Interrupt nach jedem Empfangszyklus erzeugt.

6.1.3 TX-Einheit

Im Gegensatz zur RX-Einheit besitzt die TX-Einheit ein paralleles Dateneingabesignal und eine serielle Datenausgabeleitung. Bezüglich der restlichen Schnittstelle ähneln sich beide Komponenten allerdings. Die TX-Einheit liefert ebenfalls Informationen über FIFO-Füllstand und -Überlauf und zusätzlich ein Statussignal *busy*, das angibt, ob die Einheit sich aktuell in einem Sendevorgang befindet bzw. nicht verlustfrei neue Daten entgegennehmen kann. Als Eingabe bekommt die TX-Einheit die gleichen Konfigurationsparameter übergeben, wie die RX-Einheit. Ihre Untereinheiten sind ein Schieberegister zum Senden der Daten, Pufferspeicher und ein Paritätsbitgenerator als Gegenstück zum Paritätsprüfer auf Empfängerseite. Da die TX-Einheit das Gegenstück zur RX-Einheit bildet, führt sie funktional in etwa dieselben Tätigkeiten in umgekehrter Reihenfolge aus.

6.1.3.1 Ablauf des Sendevorgangs

Wie schon bei der RX-Einheit beschrieben, verfügt auch die TX-Einheit über einen einzigen Prozess, der über eine Variable, in der die Bitposition im aktuell zu übertragenden Zeichen verwaltet wird, eine Zustandsmaschine simuliert. Zuerst werden dabei wieder die Konfigurationsparameter in Variablen geschrieben und anschließend die der jeweiligen Bitposition entsprechenden Aktionen ausgeführt. Die Bitposition 0 entspricht wieder dem Zustand *idle*, in dem eine konstante logische 1 ausgegeben wird. Ist im Pufferspeicher ein neues Datum vorhanden, so wird die Bitposition inkrementiert und die Übertragung beginnt. Bei Bitposition 1 wird zunächst ein Startbit mit einem logischen Nullpegel erzeugt, danach folgen die einzelnen Datenbits beginnend mit dem LSB[1]. Bei entsprechender Konfiguration wird anschließend das Paritätsbit und danach wiederum eine entsprechend lange Zeit ein Stoppbit erzeugt. Das Parität kann hierbei im Gegensatz zur RX-Einheit jederzeit bestimmt werden, da das Datum von Beginn der Übertragung komplett vorliegt.

6.1.3.2 Datenpufferung

Ohne einen FIFO-Speicher kann die TX-Einheit ein Zeichen zwischenspeichern. Nachdem das aktuelle Zeichen in das Schieberegister transferiert wurde, kann bereits wieder ein neues Zeichen abgelegt werden. Bei Verwendung eines FIFO-Puffers funktioniert die Dateneingabe nach demselben Prinzip mit entsprechend mehr Speicherplatz. Ist der Pufferspeicher leer, so kann der Interrupt *TXREADY* aktiviert werden, um zu signalisieren, dass der letzte Sendeauftrag abgeschlossen ist.

6.1.4 FIFO-Puffer

Als FIFO-Puffer wird eine Komponente aus dem Projekt UART16750 von Sebastian Witt [26] verwendet. Das Projekt wurde im Rahmen der Domänenanalyse für diese Arbeit betrachtet und es stellte sich heraus, dass die Komponente sich insbesondere aufgrund ihrer umfassenden Statusausgaben sehr gut für eine Wiederverwendung eignet. Die Komponente ist unter GNU LGPL lizenziert. Strukturell ist der Puffer über zwei generic-Parameter in Wortbreite und Größe konfigurierbar. Für den in dieser Arbeit entwickelten UART wird dabei allerdings nur die variable Größe genutzt und die Breite konstant als 9 angenommen, da dies die maximal benötigte Wortbreite ist und die automatisierte Codegenerierung so erleichtert wird. Intern verwaltet der Pufferspeicher eine Matrix von n mal m Bit, wobei n die Größe des Puffers und m die Wortbreite ist. Zusätzlich werden zwei Variablen als Lese- und Schreibadresszeiger verwendet. Als Steuersignale besitzt der Puffer *reset*, *clear*, *read* und *write*, als Ausgaben bietet er den Füllstand sowie Signale für die Ereignisse „voll“ und „leer“. Die Funktionalität des Puffers ist in drei Prozesse aufgeteilt, von denen der erste Prozess die Adressoperationen erledigt. Beide Adresszeiger starten bei 0 und werden bei einer Lese- oder Schreibanweisung entsprechend inkrementiert. Da beide Adressvariablen als vorzeichenlose Binärzahlen modelliert sind,

[1] niedrigstwertiges Bit

sorgt der Überlauf beim Inkrementieren automatisch für eine zyklische Adresszuweisung. Dieser Prozess ist auch für die Zuweisung des *empty*-Signals zuständig, da der Zustand „leer“ offensichtlich genau dann gegeben ist, wenn beide Adresszeiger denselben Wert haben. Der zweite Prozess sorgt für das Schreiben der Eingabedaten und die Ausgabe des aktuellen Datenwortes in Abhängigkeit von den Werten der Adresszeiger. Der dritte und letzte Prozess sorgt letztendlich für die Bestimmung der Pufferauslastung, indem er einen internen Zähler für jeden Schreibzugriff inkrementiert und beim Lesen wieder dekrementiert. Bei der Verwendung des Puffers ist besonders zu beachten, dass die Ausführung der Lese- und Schreiboperationen direkt an den Takt gebunden ist. Für eine korrekte Funktionsweise muss daher von der übergeordneten Komponente sichergestellt werden, dass die Steuersignale *read* bzw. *write* pro Zugriff nur für eine Taktperiode einen Pegel von logisch 1 haben dürfen.

6.1.5 Steuereinheit

Die Steuereinheit beinhaltet die gesamte Konfigurationslogik des UART. In einem UART mit vollem Funktionsumfang besteht die Steuereinheit somit aus vier Registern und zwei hintereinandergeschalteten Baudratengeneratoren für Receiver- und Transmittertakt sowie entsprechender Steuersignale. Die wesentlichen drei Aufgaben der Steuereineit sind die Verwaltung der Konfigurationsregister, das Erzeugen von Interrupts, die Weiterleitung der Konfigurationsdaten an die RX- und TX-Einheiten und die Erzeugung von individuellen Taktraten für diese Module. Als Eingabesignale der Steuereinheit gibt es zunächst *load*-Leitungen für die einzelnen Register. Da die UART-Ansteuerung bereits in der Top-Level-Entität übersetzt wird, muss die Steuereinheit nur noch die entsprechenden Signale und Daten an die richtigen Register weiterleiten. Zur Interrupterzeugung bekommt sie außerdem diverse Statussignale von den RX- und TX-Einheiten. Diese sind für die RX-Einheit Frameformatsfehler, Korrektheit der Parität des zuletzt empfangenen Zeichens, FIFO-Auslastung und FIFO-Überlauf und für die TX-Einheit FIFO-Überlauf und Leere des FIFO-Puffers. Ausgehend von diesen Signalen bestimmt die Steuereinheit in Verbindung mit dem aktuellen Zustand des Interrupt-Konfigurationsregisters, ob ein Interrupt ausgelöst werden muss. Die aktuelle Formatkonfiguration, die für die Kommunikation in beide Richtungen entscheidend ist, wird über einen 15 Bit breiten Vektor an die Receiver- und Transmittereinheiten ausgegeben. Die dynamische Erzeugung der Baudrate funktioniert schließlich über zwei in Serie geschaltete Baudratengeneratoren. Der erste der beiden bekommt den im Baudratenregister eingestellten Teilungsfaktor übergeben und erzeugt daraus die Baudrate für die RX-Einheit. Da die RX-Einheit ein 4-fach Sampling betreibt, muss sie vier mal so schnell getaktet werden, wie die TX-Einheit. Um nun die langsamere Taktrate für die TX-Einheit zu erzeugen, wird dem zweiten Baudratengenerator die Taktrate für die RX-Einheit als *enable* (s. u.) eingespeist und eine konstante 4 als Teilungsfaktor angelegt.

6.1.6 Baudratengenerator

Die Komponente zur Baudratenerzeugung ist im Grunde ein Zähler, der Taktflanken zählt und nach einer gegebenen Anzahl n einen Peak an der Ausgabeleitung erzeugt, wodurch effektiv die Eingabetaktrate durch den gegebenen Faktor n geteilt wird. Um dies zu tun, hat der Baudratengenerator einen Eingabetakt, eine *reset*-Leitung, einen Bitvektor zur Eingabe des Teilungsfaktors und einen *enable*-Eingang. Da die Baudrate des UART prinzipiell zu jedem beliebigen Zeitpunkt geändert werden kann, wird der Teilungsfaktor intern in einer Variable hinterlegt, die in jedem Takt aktualisiert wird. Eine interne Zählvariable zählt dann von dieser Grenze bis 0 herunter solange der Pegel der *enable*-Leitung logisch 1 ist. Erreicht die Zählvariable die 0, so wird ein Peak auf der Ausgabeleitung erzeugt. Damit dieser Aufbau eine korrekte Baudrate erzeugt, muss für die RX-Baudrate der *enable*-Eingang konstant auf 1 bleiben. Im Fall der TX-Baudrate kann die Ausgabe des ersten Generators einfach als *enable*-Signal verwendet werden.

6.1.7 Konfigurationsregister

Die verschiedenen Konfigurationsregister des UART basieren alle auf dergleichen, generischen Entität. Da die Breite der Register zur Laufzeit konstant bleibt, kann sie einfach als generic-Parameter festgelegt werden. Die Implementation des generischen Registers ist kanonisch und strukturell simpel gehalten. Das Register verfügt über eingehende und ausgehende Datenleitungen sowie die Steuersignale *load* und *reset*. Intern wird ein Bitvektor verwaltet, der die Daten speichert. Der Prozess der Entität sorgt dafür, dass die internen Daten bei einem 1-Pegel auf der *load*-Leitung mit der aktuellen Eingabe überschrieben werden. Bei einem 1-Pegel auf der *reset*-Leitung werden die internen Daten auf 0 zurückgesetzt. Der aktuelle interne Zustand wird permanent über die Datenausgänge ausgegeben.

6.1.8 Paritätsprüfung und -generierung

Die Paritätsprüfung und -generierung funktioniert grundsätzlich nach dem üblichen Prinzip der Paritätsberechnung, nämlich einer XOR-Verknüpfung aller Datenbits. Da es in diesem Fall möglich ist, zwischen gerader und ungerader Parität (odd/even parity) zu wählen, wird das Resultat der Berechnung wiederum per XOR mit dem Konfigurationsbit verknüpft. Da die Formel ohne Modifikation dem Prinzip der geraden Parität folgt und das Konfigurationsbit gemäß des Entwurfs bei der Einstellung „even parity“ logisch 0 ist, bleibt das Resultat unverändert. Im Falle der ungeraden Parität wird das Resultat noch einmal invertiert. Weil die Prüfung eines Datenwortes nur dann Sinn ergibt, wenn das Wort komplett im Speicher vorliegt, besitzt die Paritätsprüfungs-Entität eine *enable*-Leitung, deren Pegel logisch 0 ist, solange keine prüfbaren Daten vorhanden sind. Dies ist wichtig, da sonst fälschlicherweise ein Interrupt ausgelöst werden würde. Bei der Generierung eines Paritätsbits sind keine weiteren Vorkehrungen nötig, da hier offensichtlich immer ein komplettes Datenwort vorliegt.

6.2 Die Codegenerierung

In Kapitel 5 wurde bereits die Struktur des Metamodells des UART erläutert. Um nun VHDL-Dateien zu erzeugen, muss zunächst ein konkretes Modell auf Basis dieses Metamodells erstellt werden. Da nicht alle Zusammenhänge und Einschränkungen, die ein Modell erfüllen muss, allein durch die Struktur des Metamodells abgedeckt werden können, ist es erforderlich die übrigen Bedingungen in Form von Check-Constraints zu formulieren.

6.2.1 Constraints

Wie bereits angedeutet, werden viele Grundregeln für ein konkretes Modell bereits durch die Struktur des Metamodells bestimmt. Beispielsweise sorgt die Modellierung der FIFO-Speicher als Untereinheit von RX- bzw. TX-Einheit dafür, dass keine Modelle mit sinnloser Struktur, wie zum Beispiel für einen UART mit FIFO-Speicher, aber ohne eine Empfänger- oder Sendeeinheit, erstellt werden können. Auch die Begrenzung auf maximal eine RX- oder TX-Einheit ist bereits durch die Verwendung entsprechender Multiplizitäten im Metamodell fest vorgegeben. Es gibt aber auch Einschränkungen, die nicht allein durch das Metamodell ausreichend repräsentiert werden können. Insgesamt gibt es im UART-Modellierungsprojekt drei Constraints. Der erste soll dabei verhindern, dass ein UART weder mit einer TX- noch einer RX-Einheit modelliert werden kann. Der entsprechende Constraint ist in Abbildung 6.2 dargestellt.

```
context UART ERROR
    "The UART has to contain at least one of the following: RXUnit, TXUnit." :
    !(this.uartRX == null && this.uartTX == null);
```

Abbildung 6.2: Der erste Constraint des Modellierungsprojekts

Der Constraint bezieht sich auf die Gesamtstruktur des UART, weshalb das entsprechende Wurzelelement des Metamodells den Kontext bietet. Bei Verstoß gegen den Constraint muss hier offensichtlich die Codeerzeugung durch Auftreten eines Fehler abgebrochen werden, da die Struktur der erzeugten Artefakte der eigentlichen Intention des Entwurfs widersprechen würde. Der zweite Constraint des Projekts bezieht sich auf die Größe der einzelnen FIFO-Speicher. Da der Benutzer die Größe des FIFO selbst angeben kann, ist es wichtig die Eingabe entsprechend zu validieren. Es ist vorgesehen, dass es FIFO-Speicher mit einer Größe von 1 bis zu 64 Slots gibt. Da die Größe hier in Form des Exponenten einer Zweierpotenz angegeben wird, muss der eingetragene Wert offensichtlich zwischen 1 und 6 liegen. Der Constraint, der genau dies sicherstellt, ist in Abbildung 6.3 dargestellt.

Die Relevanz dieser Einschränkung liegt nur lokal beim FIFO-Speicher, weshalb dieser auch den Kontext des Constraints darstellt. Wie beim ersten Constraint muss auch hier eine Fehlermeldung erzeugt werden, da beispielsweise eine negative FIFO-Größe für die

```
context FIFO ERROR
    "FIFO size must be set and between 1 and 6." :
    this == null || ((this.size != null) && (this.size > 0) && (this.size < 7));
```

Abbildung 6.3: Der zweite Constraint des Modellierungsprojekts

Generierung von funktionsunfähigem, fehlerhaften VHDL-Code sorgen würde. Der letzte und nicht so offensichtliche Constraint des Projekts besagt, dass im Falle des Vorhandenseins von Paritätsüberprüfung und RX-Einheit auf jeden Fall auch Interrupts mit eingeschlossen werden sollten. Ist diese Kombination gewählt, können die Paritätsbits der empfangenen Daten zwar geprüft werden, aber ein Fehler könnte nie ausgegeben werden, da dies nur über Interrupts möglich ist. In diesem Fall wäre die Benutzung der Paritätsprüfung eine reine Ressourcenverschwendung. Da es sich dabei aber nur um einen Effizienzeinwand handelt und der generierte Code dennoch funktionsfähig wäre, wird im Falle eines Verstoßes nur eine Warnung ausgegeben. Der entsprechende Constraint findet sich in Abbildung 6.4.

```
context UART if (this.uartTX == null && this.parity != null) WARNING
    "Parity operation with RX should include Interrupt Support." :
    this.interrupt != null;
```

Abbildung 6.4: Der dritte Constraint des Modellierungsprojekts

6.2.2 Struktur der Templates

Ist ein konkretes Modell fertiggestellt, gilt es die Werte dieses Modells auszulesen und entsprechend der Konfiguration VHDL-Dateien zu erzeugen. Zu diesem Zweck gibt es eine Reihe von Xpand-Templatedateien, die vorgefertigte Codesegmente beinhalten, welche dann nach der gewünschten Konfiguration zusammengestellt werden.

6.2.2.1 Einstiegspunkt der Codegenerierung

Als Einstiegspunkt für den Codegenerator wird daher zunächst ein Haupttemplate benötigt, das zuerst durchlaufen werden kann und die benötigten Subtemplates aufruft oder einbindet. Um dies zu bewerkstelligen werden im Haupttemplate der Lesbarkeit halber zunächst die konkreten Konfigurationswerte des Modells ausgelesen und lokalen Variablen zugewiesen. Die Subtemplates verfügen über entsprechende Parameterlisten, die dann beim Aufruf leicht mit diesen Variablen gefüllt werden können. Im Haupttemplate werden zuerst Subtemplates für Dateien aufgerufen, die in jedem UART, ungeachtet seiner Konfiguration, aber mit jeweils unterschiedlicher interner Struktur, vorhanden sein müssen. Dazu zählt zunächst das VHDL-Package, das die Deklarationen aller internen Entitäten des UART zur Instanziierung bereitstellt. Weiterhin benötigt jeder UART die

Top-Level-Entität, eine Kontrolleinheit, einen Taktteiler zur Erzeugung der Baudrate sowie eine Version der generischen Konfigurationsregister. Je nach Modellierung kommen infolgedessen dann die optionalen Komponenten RX-Einheit, TX-Einheit, Paritätsgenerator und -prüfer sowie Interruptregister und -logik hinzu.

6.2.2.2 Hierarchie der Templatedateien

Wie in Kapitel 3 bereits erläutert, gibt es in Xpand keine eins-zu-eins-Beziehung zwischen Template- und Ausgabedateien. Durch die Möglichkeit der Definition mehrerer Templates pro Datei werden diese entkoppelt und es bietet sich dementsprechend an, die Kapselung der einzelnen Templates im Hinblick auf ihre Komplexität und Größe vorzunehmen, um eine möglichst übersichtliche und verständliche Templatestruktur zu erhalten. Abbildung 6.5 zeigt die Grobstruktur der Aufrufe der sieben Templatedateien untereinander.

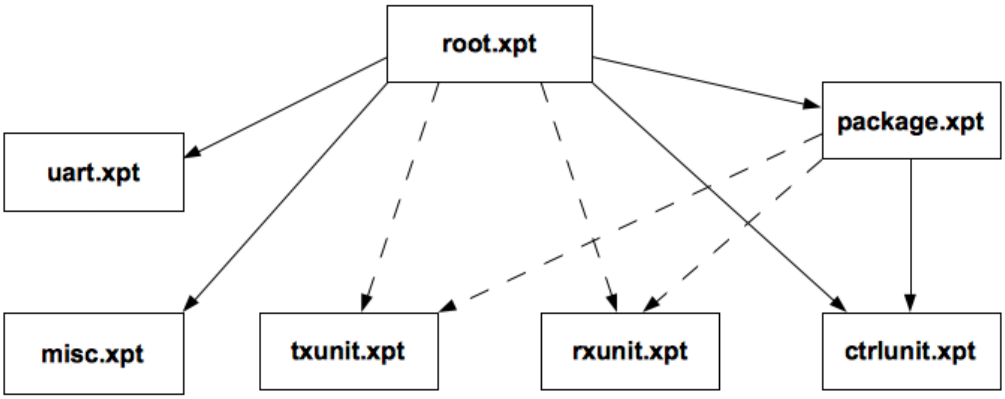

Abbildung 6.5: Grobstruktur der Templatedateien

Ein Pfeil mit durchgezogener Linie deutet hierbei einen Aufruf an, der ungeachtet der Konfiguration des UART immer auftritt, ein Pfeil mit gestrichelter Linie bedeutet, dass der Aufruf nur unter bestimmten Umständen benötigt wird. Wie bereits beschrieben startet die Bearbeitung der Templates im Haupttemplate (*root.xpt*). Da die einzelnen Templates jeweils voneinander unabhängige Dateien erzeugen, ist die Aufrufreihenfolge nicht von Bedeutung. Die Datei *uart.xpt* ist für die Erzeugung der Top-Level-Instanz des UART zuständig. Sie besteht intern aus fünf einzelnen Templates: ein Haupttemplate erzeugt die Schnittstelle nach außen sowie interne Signalzuweisungen, ein weiteres die Deklarationen der internen Signale. Die drei übrigen Templates sind für die Erzeugung der Instanzen von RX-, TX- und Steuereinheit zuständig. Die Verwendung eines eigenen Templates für die internen Signale ist hierbei dadurch begründet, dass diese sehr stark von der Konfiguration und den anderen internen Komponenten des UART abhängen und ihre Erzeugung somit verhältnismäßig komplex ist. Die Dateien *ctrlunit.xpt, rxunit.xpt* und *txunit.xpt* sind ähnlich aufgebaut und umfassen die Implementation der jeweiligen

UART-Komponenten. Die Struktur jeder dieser Komponenten ist abhängig von der Konfiguration des UART und benötigt entsprechend viele Verzweigungen im Kontrollfluss. Um ein ausreichendes Maß an Übersicht und Lesbarkeit zu behalten ist es daher nötig sie in eigene Dateien zu kapseln. Da die Schnittstellendeklarationen (port maps) der einzelnen Komponenten ebenfalls sehr variabel sind, gibt es auch hierfür eigene Templates. Da Xpand es möglich macht Templatedefinitionen aus anderen Dateien aufzurufen hat dies zudem den Vorteil, dass diese auch innerhalb des Package-Templates (*package.xpt*), wiederverwendet werden können. Die übrigen internen Komponenten des UART (Register, Taktteiler, FIFO-Speicher und Paritätsmodule) sind statisch und ihre Implementation ist unabhängig von der Konfiguration des UART nahezu immer identisch. Aus diesem Grund sind die entsprechenden Templates nicht komplex und beinhalten kaum Steueranweisungen. Die Templates dieser übrigen Komponenten sind in der Datei *misc.xpt* zu finden.

6.2.3 Extensions

Bei der Auswertung des Modells zur Anpassung des Quellcodes an die Konfiguration kommt es häufig vor, dass Zugriffe auf Modellparameter und Auswertung von Variablen sich an verschiedenen Stellen wiederholen. Zur Verbesserung der Lesbarkeit und Vermeidung von dupliziertem Code bietet es sich daher an, mehrfach vorkommende Subroutinen in Extensions zu schachteln. Das UART-Projekt kommt aufgrund der relativ einfachen Struktur des Metamodells mit nur sieben Extensions aus. Sechs dieser Extensions sind dabei einfache Wrapper für Modellzugriffe, wie Bestimmung der gewählten FIFO-Größe. Bei der siebten Extension handelt es sich um eine rekursive Funktion zur Erzeugung einer Sequenz von Nullen mit vorgegebener Länge. Diese Subroutine ist in Abbildung 6.6 dargestellt. Sie wird nur an einer Stelle zur dynamischen Generierung eines Bitvektors benötigt, ist aber wesentlich kompakter als eine Formulierung mit den reinen Sprachmitteln von Xpand.

```
String zeroes(UART this, Integer size) :
    size == 1 ? "0" : "0"+zeroes(this, size-1);
```

Abbildung 6.6: Rekursive Extension „zeroes“

6.2.4 Der Workflow

Die Workflow-Definition, die die Verwendung der einzelnen Komponenten des Projektes orchestriert, hat eine recht einfache Struktur. Da es für die Codegenerierung in diesem Projekt nicht erforderlich war eigene Generatorklassen zu entwickeln reicht es aus, Standardkomponenten ohne umfangreiche Konfiguration zu laden und zu verwenden. Zu Beginn wird das Metamodell instanziiert und das jeweils gewünschte konkrete UART-Modell geladen. Der erste Verarbeitungsschritt besteht darin, einen Check-Validator mit

der entsprechenden Constraint-Datei als Parameter zu instanziieren und das zuvor geladene Modell zu validieren. Solange hier kein Fehler auftritt wird der Standard-Xpand-Generator geladen und bekommt das Haupttemplate mit dem konkreten Modell als Eingabe. Nachbearbeitungsschritte gibt es nicht mehr, da das Xpand-Framework für VHDL keine Code-Beautifier anbietet.

6.3 Ansteuerung des UART

Die Kommunikation mit dem UART erfolgt, wie bereits beschrieben, durch das Schreiben und Lesen bestimmter Register über das Wishbone-Protokoll. Für das Betriebssystem bzw. die Treibersoftware sind diese Register über Memory Mapped I/O erreichbar, das Schreiben und Lesen funktioniert also über Zugriffe auf bestimmte Adressen. Zur Ansteuerung des UART wurde daher eine Bibliothek von C-Funktionen entwickelt, die in erster Linie dazu dienen, vom hardwarenahen, manuellen Auslesen und Schreiben der Register zu abstrahieren und die Verwendung der UART-Funktionen für den Benutzer zu erleichtern. Die Funktionen lassen sich dabei in die folgenden vier Gruppen einteilen: Übertragen von Daten, Auslesen von Daten, Konfiguration und Interruptbehandlung.

6.3.1 Senden von Daten

Die einfachste Möglichkeit Daten zum Senden an den UART zu übertragen, besteht darin, ein einzelnes Zeichen in das Transmissionsregister zu schreiben. Die Funktion `uart_putc` ermöglicht genau dies, indem sie das als Parameter übergebene Zeichen unter der entsprechenden Adresse speichert. Um zu vermeiden, dass ein noch nicht übertragenes, aber schon dem UART übermitteltes Zeichen überschrieben wird, wartet die Funktion mittels einer while-Schleife mit dem Registerzugriff bis der Status der TX-Einheit des UART auf bereit steht. Da einzelne Zeichen aber in der allermeisten Fällen nicht ausreichen, gibt es zusätzlich die Funktion `uart_puts`, die mehrere Zeichen in Form eines Character-Arrays übergeben bekommt und diese nacheinander mithilfe von `uart_putc` überträgt.

6.3.2 Auslesen empfangener Daten

Die Funktionen zum Auslesen von Daten sind nach dem gleichen Prinzip aufgebaut, wie die Funktionen zum Senden. Die Funktion `uart_getc` liest ein einzelnes Zeichen aus dem Empfangsregister, `uart_gets` liest eine bestimmte Anzahl von Zeichen, wobei die Anzahl als Parameter übergeben wird. Beim Auslesen von empfangenen Daten sind zwei Fälle zu unterscheiden. Werden Interrupts verwendet, so ist bekannt, dass der Treiber zum Zeitpunkt des Auftretens eines entsprechenden RXREADY-Interrupts mindestens soviele Zeichen abholen kann, wie der eingestellte Trigger-Level angibt, dabei allerdings mindestens eines. Es kann also vom entsprechenden Interrupt-Handler `uart_gets` mit dem bekannten Trigger-Level als Parameter aufgerufen werden. Gibt es keine Interrupts, so muss der UART per Polling regelmäßig auf neue Daten überprüft werden. Hierzu gibt

es die Funktion `uart_poll`, welche in Regelmäßigen Abständen aufgerufen werden kann, um die aktuell verfügbare Anzahl Daten abzuholen. Wenn keine Zeichen verfügbar sind, gibt die Funktion einen Nullzeiger zurück.

6.3.3 Konfiguration

Die Konfiguration des UART wird je nach Funktionsumfang über bis zu drei Register geregelt. Da jeder dieser Register (Baudrate, Format und Interrupts) eine semantisch zusammengehörige Menge an Optionen beinhaltet, existieren drei entsprechende Funktionen zum Schreiben eines Satzes von Einstellungen. Namentlich sind das die Funktionen `uart_set_br`, `uart_set_fmt` und `uart_set_int`. Die erste Funktion, `uart_set_br`, erwartet die Taktfrequenz in Hz und die gewünschte Baudrate als Parameter. Sie berechnet intern den benötigten Teilungsfaktor und schreibt ihn in das Baudratenregister. Da die anderen beiden Register im Gegensatz zum Baudratenregister mehr als nur einen logischen Wert beinhalten, gibt es zwei Datenstrukturen, in die die Optionen übersichtlich eingetragen werden können. Die mit Werten belegten Datenstrukturen werden dann den Funktionen übergeben, welche die Eingabe in einen Bitvektor umwandeln und in die richtigen Register schreiben.

6.3.4 Interruptbehandlung

Das Interruptsystem basiert auf dem globalen UART-Interrupt-Handler (`uart_handle_irq`), der beim Auftreten eines UART-Interrupts vom globalen Interrupt-Handler aufgerufen wird. Innerhalb von `uart_handle_irq` wird zunächst der Interruptstatusvektor des UART gelesen und dann sequenziell auf einen Interrupt nach dem anderen geprüft. Durch die Reihenfolge der Überprüfung wird hier analog zum globalen Interrupt-Handler die Priorität der einzelnen Interrupts bestimmt, indem ein Interrupt, auf den früher geprüft wird, alle unter ihm verdeckt. Je nach aufgetretenem Interrupt wird dann eine der folgenden Funktionen aufgerufen:

- `uart_handle_rx_ready`
- `uart_handle_tx_ready`
- `uart_handle_rx_of`
- `uart_handle_tx_of`
- `uart_handle_par_err`
- `uart_handle_frm_err`

Während für die Funktion `uart_handle_rx_ready` das Auslesen der empfangenen Daten naheliegt, hängen die Aktionen der anderen Handler im wesentlichen vom konkreten Anwendungsfall und vom Benutzer ab. Dementsprechend ist diese Funktion im Gegensatz zu den übrigen bereits vorimplementiert. Für die Funktionen `uart_handle_par_err` und

`uart_handle_frm_err` ist weiterhin wichtig, dass das von einem Fehler betroffene Zeichen ausgelesen wird, da diese Interrupt-Signale im UART nur durch einen Lesezugriff unterbunden werden.

7 Evaluation

Dieses Kapitel umfasst die Evaluation des entwickelten UART und der Codeerzeugung mithilfe des Xpand-Frameworks. Im ersten Teil des Kapitels wird zunächst thematisiert, wie der Test der entwickelten Komponenten angesichts der vielen verschiedenen Konfigurationen umgesetzt wurde. Im zweiten Teil wird anschließend eine Auswertung der Ressourcenausnutzung des UARTs in Abhängigkeit von verschiedenen Konfigurationen vorgenommen. Da im Verlauf dieser Arbeit zum ersten Mal ausschließlich Xpand zur Codegenerierung verwendet wurde, bildet den Abschluss des Kapitels ein Vergleich der Templatesprachen XVCL und Xpand in Verbindung mit einer Beurteilung der Eignung von Xpand für diese Anwendungszweck.

7.1 Methodik des UART-Tests

Das Testen von hochkonfigurierbaren Systemen wird durch die große Anzahl verschiedener Testkonfigurationen erheblich erschwert. Für einen erschöpfenden Test der UART-Familie müsste man zuerst alle denkbaren Varianten des UART generieren, synthetisieren und anschließend auf dem FPGA alle Konfigurationsmöglichkeiten testen. Dieser Aufwand kann allerdings stark minimiert werden, wenn man die Zusammensetzung der einzelnen UART-Varianten genauer betrachtet und ausnutzt, dass viele Teile aus denselben Codebausteinen aufgebaut sind oder unabhängig voneinander arbeiten und daher nicht mehrfach getestet werden müssen. Darunter fallen zum einen die RX- und TX-Einheiten und deren Teilmodule, die als solches nicht miteinander interagieren, sondern nur individuell von der Steuereinheit angesprochen werden. Ebenfalls kann es umgangen werden, Systeme mit verschiedenen FIFO-Größen zu überprüfen, da sich die Schnittstellen nach außen nicht unterscheiden und die unterschiedlichen Größen des Puffers nur durch Mehrfachinstanziierung gleicher Elemente realisiert werden. Statt also alle möglichen Zusammensetzungen zu testen, kann beim UART-Test das Ziel verfolgt werden eine Auswahl von Testsystemen aufzubauen, die gemeinsam eine vollständige Codeabdeckung liefern. Die Laufzeitkonfiguration der UART-Funktionen kann dann auf diesen repräsentativen Testsystemen verifiziert werden. Um das Auffinden von Fehlern zu vereinfachen, wurden die einzelnen Komponenten mithilfe entsprechender Testbenches[1] im Simulator verifiziert, sodass der Abschlusstest letztendlich nur noch darauf abzielt, Fehler bei der Interaktion der Einzelmodule zu finden.

[1] Nach außen abgeschlossene VHDL-Entities mit internen Stimuli

7.1.1 Auswahl der Testsysteme

Da ein UART als Einzelkomponente nicht in einer Realumgebung testbar ist, muss zunächst ein MPSoC erstellt werden, in das der UART eingebettet werden kann. Für diese Testserie wurde ein funktionierendes MPSoC aus einem anderen Projekt ausgewählt, in dem der ursprüngliche rudimentäre LavA-UART verwendet wurde. Das MPSoC besteht aus vier Untersystemen, von denen das erste über den UART verfügt. Zum Test wurden auf der Grundlage dieses Systems acht Varianten von UARTs generiert und in die Systemstruktur eingebunden. Diese Varianten unterscheiden sich, wie in Abbildung 7.1 dargestellt, durch die Merkmale FIFO-Puffer, Interrupts und Paritäten.

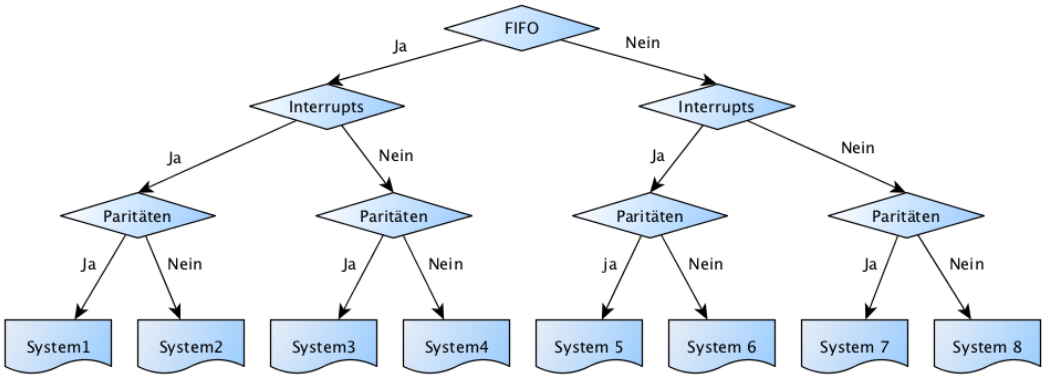

Abbildung 7.1: Unterscheidung der Testsysteme

Eine weitere Zusammenlegung der Testfälle ist aufgrund der jeweils unterschiedlichen Kompositionsstruktur in diesem Fall nicht mehr ohne Einbußen in der Codeabdeckung möglich.

7.1.2 Ablauf des Laufzeittests

Um die einzelnen Varianten zu testen, müssen sie zunächst synthetisiert und auf ein FPGA gebracht werden. In diesem Fall handelte es sich um ein FPGA-Board der Firma Digilent mit einem Xilinx Virtex-5 XC5VLX110T FPGA [29]. Zur Ansteuerung des UART über die parallele Schnittstelle wurden die im vorigen Kapitel beschriebenen Ansteuerungsfunktionen verwendet, zur Ansteuerung über die serielle Schnittstelle wurde dahingegen auf einem mit dem Board verbundenen Rechner das serielle Terminal HTerm verwendet. Bei allen Testsystemen wurden zunächst die einfachen Funktionalitäten Senden und Empfangen über programmierte Ausgaben bzw. manuelle Eingaben über das Terminal überprüft. Dabei wurde zunächst die Standardeinstellung von 8 Datenbits ohne Parität bei einer Baudrate von 115200 Bit pro Sekunde verwendet. Im Anschluss wurden die immer vorhandenen Konfigurationsoptionen wie Stoppbitanzahl, Datenwortbreite und Baudrate über ein entsprechendes Programm für das MPSoC variiert und wiederum die Aus- und Eingaben am Terminal verifiziert. Das Testen der Interrupts wurde

für jeden Interrupt individuell durchgeführt. Der Interrupt *RXREADY* wurde durch manuelle Eingabe der je nach System nötigen Menge von Zeichen und Ausgabe einer Testsequenz manuell verifiziert. Für den Interrupt *TXREADY* wurde eine Testsequenz ausgegeben und bei Eintreten des Interrupts eine weitere erzeugt. Paritätsfehler konnten manuell über Senden von Zeichen mit falschem Paritätsmodus, also einem umgekehrten Paritätsbit ausgelöst werden. Auf ähnliche Weise konnten auch Fehler im Frameformat über das Senden von Zeichen mit Paritätsbit ohne Einstellung einer Parität am UART provoziert werden. Das Eintreten der Interrupts für Pufferüberläufe der TX- und RX-Einheiten wurde schließlich durch ununterbrochenes Senden von Daten an den UART bzw. durch das Nichtabholen empfangener Zeichen verifiziert.

7.2 Evaluation des UART

Zur Beurteilung des entwickelten UARTs sollen an dieser Stelle der Ressourcenbedarf des Moduls herangezogen werden. Da es eines der Ziele war, durch die Option des Auslassens einzelner Komponenten Ressourcen zu sparen und somit die Ressourcenbelegung des UART auf dem FPGA mit seinem Funktionsumfang skalieren zu lassen, wird zunächst eine entsprechende Analyse basierend auf der Implementation für ein Xilinx Virtex-5 FPGA präsentiert. Zur Beurteilung der Skalierung und Betrachtung der allgemeinen Ressourcenauslastung des UART wurden drei verschiedene Varianten mit unterschiedlichen Funktionsumfängen generiert. Eine minimalistische Variante, die nur aus einer Sendeeinheit ohne FIFO-Puffer und den übrigen verpflichtenden Komponenten zur Konfiguration besteht. Weiterhin eine Variante mit einer Teilmenge der möglichen Komponenten, bestehend aus RX- und TX-Einheiten mit Interruptunterstützung und kleinem Pufferspeicher, letztlich eine Variante, die den maximal möglichen Funktionsumfang beinhaltet.

7.2.1 Minimalkonfiguration

Die kleinste modellierte UART-Variante lieferte nach Synthese und Implementation folgende Ressourcenbelegung:

Modul	**Slices**	**Slice Reg**	**LUTs**
Top-Level	6	0	6
Steuereinheit	40	113	91
TX-Einheit	18	24	24
Gesamt	64	137	121

Tabelle 7.1: Ressourcenbedarf der minimalen UART-Variante

7.2.2 Teilmenge der möglichen Komponenten

Die zweite UART-Variante bestand aus RX- und TX-Einheit mit je 16-Slot-FIFO-Speichern und der Möglichkeit Interrupts zu verwenden. Die Ergebnisse der Synthese sind in folgender Tabelle zusammengefasst:

Modul	**Slices**	**Slice Reg**	**LUTs**
Top-Level	10	0	10
Steuereinheit	51	123	99
RX Top	31	26	34
RX FIFO	74	168	83
TX Top	18	25	24
TX FIFO	73	168	82
Gesamt	257	510	332

Tabelle 7.2: Ressourcenbedarf der mittelgroßen UART-Variante

7.2.3 Kompletter Funktionsumfang

Die letzte betrachtete UART-Variante verfügte über den größtmöglichen Funktionsumfang und jeweils 64 FIFO-Slots für beide Einheiten. Die Untersuchung lieferte folgende Ergebnisse:

Modul	**Slices**	**Slice Reg**	**LUTs**
Top-Level	10	0	10
Steuereinheit	55	125	102
RX Top	35	28	39
RX FIFO	232	606	290
Parity Checker	3	1	3
TX Top	22	25	36
TX FIFO	236	606	290
Parity Gen	1	1	2
Gesamt	594	1392	772

Tabelle 7.3: Ressourcenbedarf der kompletten UART-Variante

7.2.4 Auswertung der Ergebnisse

Setzt man die Einzelbetrachtungen in Beziehung, so ergibt sich das in Abbildung 7.2 dargestellte Ergebnis. Insgesamt ist zu erkennen, dass die Ressourcenanforderungen des

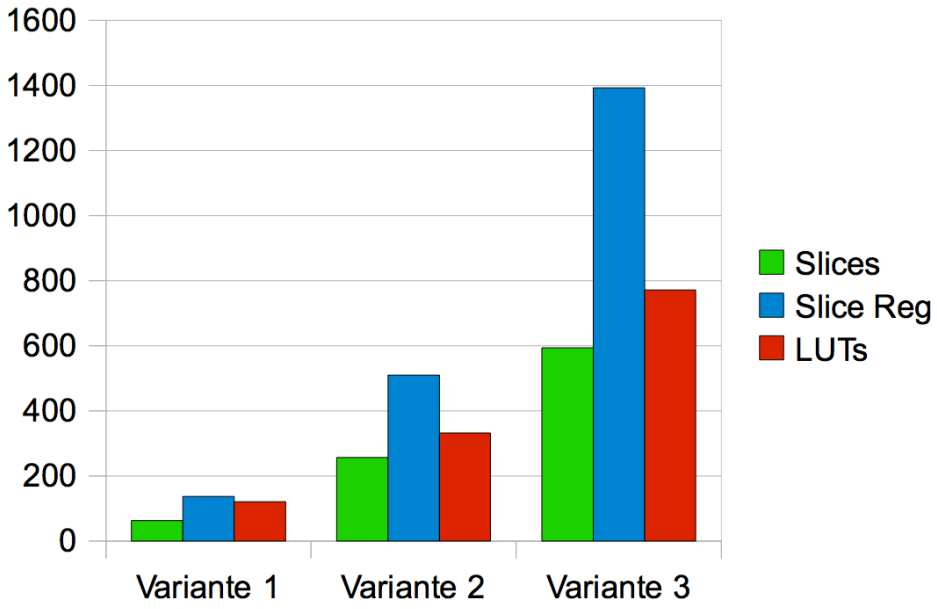

Abbildung 7.2: Gegenüberstellung der Ergebnisse

UART in Abhängigkeit vom gewählten Funktionsumfang über ein breites Spektrum skalieren. Bei genauerer Betrachtung der Einzelergebnisse zeigt sich, dass die Ressourcenbelegung hauptsächlich in Relation zur gewählten FIFO-Größe steht. Dies war zu erwarten, da es sich beim FIFO-Puffer sowohl um die größte Speicherstruktur als auch um eine der schaltungstechnisch komplexeren Komponenten handelt. Die übrigen Komponenten skalieren weniger stark, aber ebenfalls erwartungsgemäß mit der Anzahl vorhandener Module im UART.

7.3 Evaluation der Codegenerierung

Nachdem die Implementierung des UART analysiert wurde, soll nun das Generierungsprojekt als der zweite große Implementationsabschnitt dieser Arbeit evaluiert werden. Dazu wird ein Vergleich zwischen Xpand und der im LavA-Projekt bisher verwendeten Sprache XVCL angestellt, auf dem dann die Einschätzung der Eignung von Xpand als alleiniges Codegenerierungswerkzeug aufbaut.

7.3.1 Vergleich von Xpand und XVCL

Für den Vergleich der beiden Templatesprachen werden hier als Vergleichskriterien Struktur, Ausdrücksstärke und angebotene Werkzeugunterstützung herangezogen. Da der Ver-

gleich dazu dienen soll zu entscheiden, welche der beiden besser für die Verwendung im LavA-Workflow geeignet ist, bieten sich diese Kriterien an, da sie die Benutzerfreundlichkeit im täglichen Arbeitsablauf widerspiegeln.

7.3.1.1 Struktur und Lesbarkeit der Templates

Auf den ersten Blick unterscheiden sich Xpand-Templates und x-frames in ihrer Struktur nur wenig, da beide Formate aus mit Steueranweisungen durchzogenem Quelltext bestehen. Unter der Voraussetzung, dass beide Dateitypen mit Syntax-Highlighting betrachtet werden, ähnelt sich die grundsätzliche Lesbarkeit der einzelnen Dateien ebenfalls. Die Übersichtlichkeit der XML-Dateien von XVCL ist dabei aufgrund des typisch großen Anteils an spitzen Klammern und Anführungszeichen etwas schlechter. Im Gegenzug ist man bei Xpand für Syntax-Highlighting an den schlichten Editor des Eclipse-Plugins gebunden, da die Templates in einer eigenen DSL[2] verfasst werden, wohingegen XVCL in jedem gängigen XML-Editor angezeigt werden kann. Während XVCL für große Dateien insgesamt relativ unübersichtlich bleibt, hat der Benutzer bei der Verwendung von Xpand den Vorteil, dass es mehr Möglichkeiten zur Kapselung und Komplexitätsreduktion gibt. Zum einen können komplexe Ausdrücke in Extensions ausgelagert werden, zum anderen können auch mehrere Untertemplates mit beliebigen Parameterlisten erstellt werden, wodurch insgesamt für geübte Programmierer ein intuitiverer Entwurf möglich ist, als mit dem deklarativen Stil von XVCL. Ein deutlicher Vorteil von XVCL gegenüber Xpand ist jedoch die Kontrolle über Generierung von Whitespace. Diese ist in Xpand schwieriger, da hier explizit nach jeder Steueranweisung die Erzeugung einer Freizeile durch Anhängen eines Minuszeichens unterdrückt werden muss. Diese umständliche Handhabung rührt daher, dass zur Formatierung der Ausgabedateien die Verwendung eines entsprechenden Code-Beautifiers angedacht ist. Ein solcher existiert leider bisher für VHDL noch nicht, weshalb bei der Verwendung von Xpand ein Kompromiss zwischen der Qualität der Templates und der Ausgabedateien getroffen werden muss.

7.3.1.2 Mächtigkeit der Sprachelemente

Im Vergleich zu XVCL bietet Xpand wesentlich mächtigere Sprachkonstrukte an. Während der Grundumfang beider Sprachen nur Variablenzuweisungen und -auswertungen sowie while- und foreach-Schleifen umfasst, verfügt Xpand mithilfe von Expressions und Extensions über eine ähnliche Mächtigkeit wie Java. Darüberhinaus kann Xpand ohne weiteren Aufwand auf grafisch erstellten Modellen arbeiten und leicht auf dort vorhandene Attribute zugreifen ohne diese vorher in Textform zu bringen, was insgesamt zu einer wesentlich geringeren Zahl lokaler Variablen führt und die Übersichtlichkeit der Templates positiv beeinflusst.

[2]Domain-specific language

7.3.1.3 Werkzeugunterstützung

Sowohl XVCL als auch Xpand bieten ein Eclipse-Plugin als Arbeitsumgebung an, setzen dabei aber unterschiedliche Schwerpunkte. Zwar bieten beide Editoren Quellcode-Highlighting und Auto-Vervollständigung, aber während die XVCL-Workbench eher auf die Automatisierung der Frameentwicklung und Refactoring ausgerichtet ist, bietet die Xpand-Arbeitsumgebung eine Schnittstelle zu anderen Modellierungselementen der Eclipse MWE und versucht den Codegenerierungs-Workflow sowie das Zusammenspiel der einzelnen Komponenten zu erleichtern.

7.3.2 Eignung von Xpand

Die wesentlichen Kritikpunkte an XVCL sind zum einen die schlechte Lesbarkeit der x-Frames, die deutlich nachteilige Auswirkungen auf die Wartbarkeit des Quellcodes hat, zum anderen aber auch die eingeschränkte Ausdrucksstärke der Sprache, die dazu führt, dass oft sehr umständliche Konstrukte entwickelt werden müssen, um verhältnismäßig einfache Ausdrücke zu formulieren. Erschwerend kommen außerdem inhärente Probleme von Standard-XML, wie die fehlende Typisierung von Variablen oder auch die Kodierung von Sonderzeichen hinzu. Xpand hat hier gewisse Vorteile, besitzt aber bezüglich der Formatierung von Templates und Ausgabedateien ebenfalls einige Nachteile. Gerade angesichts der Tatsache, dass die aktuelle Variante, in der die Generierung eines SPC für XVCL durch Xpand übernommen wird, aber einen zusätzlichen Aufwand bedeutet und XVCL keine Funktionen bietet, die Xpand nicht auch bereitstellt, ist die Ersetzung von XVCL durch einen vollständigen Xpand-Workflow in meinen Augen durchaus eine praktikable Alternative.

8 Fazit und Ausblick

Dieses Kapitel bildet den Abschluss dieser Arbeit und gibt zu diesem Zweck einen letzten Überblick über die im Verlauf der Arbeit entwickelten Komponenten und gewonnenen Erkenntnisse. Im zweiten Teil des Kapitels wird schlussendlich ein Ausblick bezüglich eventueller Erweiterungsmöglichkeiten der entwickelten UART-Familie gegeben.

8.1 Zusammenfassung

Im Laufe dieser Bachelorarbeit wurde im Kontext des LavA-Projekts ausgehend von einer einfachen, statischen VHDL-Implementierung eines UART eine dynamische UART-Familie entwickelt, aus welcher mithilfe des Codegenerierungsframeworks Xpand und der Eclipse Modeling Workflow Engine verschiedene zur Laufzeit konfigurierbare UART-Varianten maßgeschneidert werden können. Zu diesem Zweck wurde in Kapitel 4 hinsichtlich der möglichen Konfigurationsmöglichkeiten zunächst eine Domänenanalye durchgeführt, auf deren Grundlage der Entwurf des UART in Kapitel 5 umgesetzt werden konnte. Hier wurden sowohl UARTs industriell vertriebener Mikrocontroller, als auch verschiedene Open-Source-Entwicklungen betrachtet, um eine sinnvolle Auswahl zu realisierender Funktionen zu gewinnen. In Kapitel 5 wurden die Konfigurationspunkte anschließend Konfigurationszeitpunkten (Laufzeit und Modellierungszeit) zugeordnet, wobei die Konfigurierung zur Modellierungszeit dem Zweck dient, durch das Auslassen nicht benötigter Komponenten Ressourcen einzusparen. Aus dieser Aufteilung wurde zunächst ein Metamodell für die Maßschneiderung von UARTs innerhalb des Eclipse-Projekts entworfen. Anschließend wurden verschiedene VHDL-Module abgeleitet und für den Verbund dieser ein Blockschaltbild als Grundlage der Implementation entworfen. Die drei wesentlichen interagierenden Module des UART sind die RX-Einheit (Empfänger), die TX-Einheit (Sender) und die koordinierende Steuereinheit, die die Konfigurationsparameter sowie Ein- und Ausgaben verwaltet. Die fertiggestellten VHDL-Module wurden dann in Xpand-Templates eingebettet und erweitert um Constraints und Extensions dem Generierungsprozess innerhalb des Eclipse-Projekts zugänglich gemacht. Bei der Modellierung eines UART mit dem entstandenen Projekt ist es so möglich unterschiedliche Varianten mit Sender und/oder Empfänger zu erstellen. Außerdem können bei Bedarf FIFO-Puffer, Interruptfunktionen oder Paritätsunterstützung hinzukonfiguriert werden. Im Betrieb können die Baudrate, Kommunikationsformat und Interruptaktivierung geändert werden. Nach Abschluss der Implementierung wurde das entwickelte System in Kapitel 7 analysiert und die Ergebnisse bewertet. Die Analyse des Ressourcenverbrauchs ergab, dass der implementierte UART durchaus merklich mit dem jeweils gewählten Funktionsumfang skaliert. Letztlich war die Frage zu klären, ob es sinnvoll

ist, den aktuellen LavA-Workflow dahingehend zu modifizieren, dass XVCL vollständig durch Komponenten des Xpand-Frameworks ersetzt werden kann. Nach Abwägung der Vor- und Nachteile beider Sprachen zeigte sich, dass die Ablösung von XVCL durch Xpand durchaus sinnvoll ist. Dies begründet sich in erster Linie darin, dass Xpand Vorteile bezüglich Übersichtlichkeit und Ausdrucksstärke besitzt und die Nutzung von XVCL wegen der zusätzlichen Transformation des grafischen Modells in ein textuelles einen Mehraufwand darstellt, der nicht durch zusätzliche Fähigkeiten seitens XVCL gerechtfertigt wird.

8.2 Ausblick

Ausgehend vom Umfang der Implementation zum Ende dieser Arbeit gibt es noch einige denkbare Erweiterungen. Eine mögliche Erweiterung für den Funktionsumfang des UART wäre dabei zunächst die Implementation eines synchronen Kommunikationsmodus als Alternative zur asynchronen Kommunikation, über den modernere UART größtenteils verfügen. Bezüglich des Generierungsprojekts wäre es für den Benutzer komfortabel direkt passende Software zur Ansteuerung des UART zu generieren, anstatt eine Bibliothek von Funktionen anzubieten. Viele Funktionen, insbesondere zum Lesen und Schreiben von Daten, beinhalten viele Fallunterscheidungen, um die verschiedenen UART-Varianten über eine gleichförmige Schnittstelle ansprechen zu können. Würde man über Templates individuelle Funktionen erzeugen, wären diese insgesamt kompakter. Eine im Verhältnis zum Nutzen sehr aufwendige aber nichtsdestotrotz interessante Erweiterung wäre die Entwicklung eines VHDL-Code-Beautifiers für Xpand-Codegeneratoren. So könnte der momentan nötige Kompromiss bezüglich Lesbarkeit und Wartbarkeit zwischen Templates und generiertem Code eliminiert werden.

Literaturverzeichnis

[1] Marwedel, Peter: *Embedded System Design - Embedded Systems Foundations of Cyber-Physical Systems*. Springer, 2010

[2] TU Dortmund – Lehrstuhl 12, Arbeitsgruppe Eingebette Systemsoftware: *LavA: Laufzeitplattform für anwendungsspezifische verteilte Architekturen*. http://ess.cs.tu-dortmund.de/DE/Research/Projects/LavA/

[3] Stahl, Thomas ; Völter, Markus ; Efftinge, Sven ; Haase, Arno: *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. dpunkt.verlag, 2007

[4] Lohmann, Daniel ; Hofer, Wanja ; Schröder-Preikschat, Wolfgang ; Streicher, Jochen ; Spinczyk, Olaf: CiAO: an aspect-oriented operating-system family for resource-constrained embedded systems. In: *Proceedings of the 2009 conference on USENIX Annual technical conference*. Berkeley, CA, USA : USENIX Association, 2009 (USENIX'09), 16–16

[5] Wikipedia: *Universal Asynchronous Receiver Transmitter*. http://de.wikipedia.org/wiki/UART. Version: Mai 2011

[6] The Eclipse Foundation: *Xpand Project*. http://www.eclipse.org/modeling/m2t/?project=xpand

[7] openArchitectureWare.org: *openArchitectureWare User Guide – Xpand reference*. http://www.openarchitectureware.org/pub/documentation/4.3.1/html/contents/core_reference.html#xpand_reference_introduction

[8] Jarzabek, Stan ; Bassett, Paul ; Zhang, Hongyu ; Zhang, Weishan: XVCL: XML-based variant configuration language. In: *Proceedings of the 25th International Conference on Software Engineering*. Washington, DC, USA : IEEE Computer Society, 2003 (ICSE '03). – ISBN 0–7695–1877–X, 810–811

[9] Böckle, G. ; Knauber, P. ; K.Pohl ; Schmid, K. (Hrsg.): *Software–Produktlinien – Methoden, Einführung und Praxis*. First. dpunkt.Verlag, 2004

[10] Czarnecki, Krzysztof ; Eisenecker, Ulrich W.: *Generative Programming – Methods, Tools and Applications*. Addison–Wesley, 2000

[11] National Semiconductor Corporation: *PC16550D Universal Asynchronous Receiver/Transmitter with FIFOs*. http://www.national.com/ds/PC/PC16550D.pdf. Version: 1995

[12] Meier, M. ; Engel, M. ; Steinkamp, M. ; Spinczyk, O.: LavA: An Open Platform for Rapid Prototyping of MPSoCs. In: *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, 2010. – ISSN 1946–1488, S. 452 –457

[13] National University of Singapore: *XML-based Variant Configuration Language Project Site*. http://xvcl.comp.nus.edu.sg/overview_brochure.php

[14] The Eclipse Foundation: *Eclipse Modeling Framework*. http://www.eclipse.org/modeling/emf/

[15] openArchitectureWare.org: *Check Language reference*. http://www.openarchitectureware.org/pub/documentation/4.3.1/html/contents/core_reference.html#Check_language

[16] opencores.org: *Specification for the WISHBONE System–on–Chip (SoC) Interconnection Architecture for Portable IP Cores; Revision B3*. cdn.opencores.org/downloads/wbspec_b3.pdf. Version: September 2002

[17] Kang, K. ; Cohen, S. ; Hess, J. ; Nowak, W. ; Peterson, S.: Feature–oriented Domain Analysis (FODA): Feasibility Study / Pittsburgh, PA: Carnegie Mellon University, Software Engineering Institute. 1990. – Forschungsbericht

[18] Spinczyk, Olaf: *Software–Produktlinien*. http://ess.cs.tu-dortmund.de/Teaching/SS2010/SuS/Downloads/03.3-SW-Produktlinien.pdf. Version: 2010

[19] The Eclipse Foundation: *Xpand Documentation - Expressions*. http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.xpand.doc/help/ch01s06.html. Version: 2011

[20] Object Management Group: *Object Constraint Language – Version 2.3*. http://www.omg.org/spec/OCL/2.3/Beta2/PDF. Version: 2010

[21] The Eclipse Foundation: *Xpand Documentation - Xtend*. http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.xpand.doc/help/ch01s06.html. Version: 2011

[22] Atmel Corporation: *ATMEL ATmega32(L) complete datasheet*. http://www.atmel.com/dyn/resources/prod_documents/doc2503.pdf. Version: 2011

[23] NXP Semiconductors: *LPC2377/78 product datasheet*. http://www.nxp.com/documents/data_sheet/LPC2377_78.pdf. Version: 2010

[24] NXP Semiconductors: *LPC23XX User manual*. http://www.keil.com/dd/docs/datashts/philips/lpc23xx_um.pdf. Version: 2009

[25] Microchip Technology Inc.: *Microchip PIC32 Family Reference Manual – Section 21: UART*. http://ww1.microchip.com/downloads/en/DeviceDoc/61107F.pdf. Version: 2010

[26] Witt, Sebastian: *UART16750*. http://www.opencores.org/project,uart16750. Version: 2011

[27] Gorban, Jacob ; Mohor, Igor ; Markovic, Tadej: *UART 16550 core*. http://www.opencores.org/project,uart16550. Version: 2010

[28] Wikipedia: *RS-232*. http://de.wikipedia.org/wiki/RS-232. Version: August 2011

[29] Digilent Inc.: *Digilent Virtex-5 OpenSPARC Evaluation Platform*. http://www.digilentinc.com/Products/Detail.cfm?NavTop=2&NavSub=599&Prod=XUPV5. Version: August 2011

Abbildungsverzeichnis

2.1 Beispiel für ein x-framework [12] 7
2.2 Das Hardware-Metamodell in LavA 8
2.3 Die Hardware-Konfiguration der LavA-Plattform [2] 9
2.4 Der Aufbau der ursprünglichen LavA UART-Komponente 10
2.5 Beispiel eines Merkmaldiagramms. Quelle [18] 11

3.1 Beispiel für einfache Expressions 13
3.2 Beispiel für eine rekursive Extension 15
3.3 Beispiel für einen einfachen Constraint 16
3.4 Beispiel für eine workflow-Datei 17
3.5 Ein einfaches Generics-Beispiel. Links die entity-Deklaration, rechts die Instanziierung. 18
3.6 Die Syntax des generate–Statements in VHDL 18

4.1 Merkmalmodell für einen UART 25
4.2 Subdiagramm für kommunikationsbezogene Merkmale 26
4.3 Subdiagramm für Interrupt-bezogene Mermale 27

5.1 Klassendiagramm des Ecore-Metamodells 32
5.2 Blockdiagramm des UART 36

6.1 Timing-Schema nach RS-232. Quelle: [28] 39
6.2 Der erste Constraint des Modellierungsprojekts 43
6.3 Der zweite Constraint des Modellierungsprojekts 44
6.4 Der dritte Constraint des Modellierungsprojekts 44
6.5 Grobstruktur der Templatedateien 45
6.6 Rekursive Extension „zeroes“ 46

7.1 Unterscheidung der Testsysteme 52
7.2 Gegenüberstellung der Ergebnisse 55

Tabellenverzeichnis

5.1 Zuordnung der Konfigurationspunkte 31
5.2 Mapping des Format-Konfigurationsregisters 33
5.3 Mapping der Interrupt-Register 34
5.4 Mapping der Speicheradressen des UART 35

7.1 Ressourcenbedarf der minimalen UART-Variante 53
7.2 Ressourcenbedarf der mittelgroßen UART-Variante 54
7.3 Ressourcenbedarf der kompletten UART-Variante 54