technische universität
dortmund

Diplomarbeit

Maßschneiderung von MPSoCs durch Code-Analyse

Matthias Steinkamp
22. September 2010

Betreuer:
Prof. Dr.-Ing. Olaf Spinczyk
Prof. Dr.-Ing. Christian Wietfeld

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 22. September 2010

Matthias Steinkamp

Zusammenfassung

Die Anwendungen eingebetteter Systeme werden immer komplexer. Das modernste Beispiel sind hochauflösende Fernsehgeräte mit zunehmender Unterstützung für eine dreidimensionale Darstellung.
Der Bedarf an Rechenleistung der Hardware steigt dementsprechend. Wie vom Desktop-Bereich bekannt ist, werden die Taktfrequenzen der Prozessoren, auf Grund von physikalischen Grenzen, nicht weiter gesteigert, sondern mehrere Rechenkerne in einem Chip verbaut. Die Anwendungen müssen parallelisiert werden, um von dieser Entwicklung zu profitieren.
Zusätzlich lassen sich zwei weitere Treds erkennen: Zum einen wächst die Komplexität integrierter Schaltungen dank einer steigenden Integrationsdichte. Auch rekonfigurierbare Hardware, sog. FPGAs, werden dadurch Leistungsstärker. Zum anderen lässt sich immer mehr Hardware durch Beschreibungssprachen entwerfen. Daraus resultierend sind selbst Multiprozessorsysteme textuell konstruierbar, die anschließend auf einem FPGA realisiert werden können.
In dieser Diplomarbeit soll eine Technik vorgestellt werden, welche die von einer Anwendung benötigte Hardware maßgeschneidert zur Verfügung stellt. Damit wird eine durchgängige Konfigurierung und Entwicklung von Hard- und Software ermöglicht. Basierend auf einer konfigurierbaren Hardwareplattform und einer umfangreichen Werkzeugkette, wird dem Entwickler jegliche Arbeit mit Hardwarebeschreibungssprachen erspart.
Als eine alternative Zielarchitektur wird ein Simulator vorgestellt, der die gesamte Plattform auf dem Betriebssystem Linux abspielen kann. Fehlerquellen oder beschränkte Ressourcen werden hier ausgeblendet, sodass der Fokus einzig bei der Anwendungsentwicklung liegt.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Das LavA-Projekt . . . 1
1.2 Ziele . . . 3
1.3 Gliederung der Arbeit . . . 3

2 Grundlagen **5**
2.1 Modellgetriebene Softwareentwicklung mit openArchitectureWare . . . 5
2.2 Puma . . . 7
2.3 Die konfigurierbare Hardwareplattform . . . 7
2.4 Related Work . . . 8
2.4.1 Daedalus . . . 8
2.4.2 Koski . . . 10

3 Konzeption der Werkzeugkette **13**
3.1 Repräsentation von Hardware in Software . . . 13
3.1.1 C++ Templates . . . 14
3.1.2 Aufruf von statischen Methoden . . . 15
3.1.3 Typdefinitionen . . . 16
3.1.4 Globale Objekte . . . 17
3.2 Die Hardware API . . . 18
3.2.1 Modellgetriebene Entwicklung der Hardware API . . . 18
3.2.2 Instanziierung der Hardware API . . . 24
3.2.3 Unterschiedliche Zielarchitekturen . . . 25
3.3 Aufbau der Werkzeugkette . . . 26
3.3.1 Die Eingabe der Werkzeugkette . . . 26
3.3.2 Analyse der Anwendung . . . 27
3.3.3 Ausführung auf einer Zielplattform . . . 27

4 Entwurf ausgewählter Werkzeuge **31**
4.1 Der Parser . . . 31
4.1.1 Suche und Analyse der Hardware API . . . 32
4.1.2 Datenhaltung der Instanzen . . . 37
4.1.3 Manipulation der Anwendung . . . 39
4.1.4 Modellgenerierung . . . 41
4.2 Der Simulator . . . 42
4.2.1 Abbildung des MPSoCs auf Linux . . . 42

4.2.2 Vernetzung durch den Simulator . . . 44
4.3 Ressourcenbedarf von MPSoCs . . . 45
4.3.1 Definition der Kosten eines MPSoCs . . . 45
4.3.2 Integration in die Werkzeugkette . . . 46

5 Realisierung ausgewählter Werkzeuge 47
5.1 Der Parser . . . 47
5.1.1 Umfang der Implementierung . . . 47
5.1.2 Suche nach globalen Objekten . . . 48
5.1.3 Schwierigkeiten bei der Entwicklung . . . 49
5.2 Der Simulator . . . 50
5.2.1 Umfang der Implementierung . . . 50
5.2.2 Die Umsetzung der IPC Controller . . . 50
5.3 Berechnung des Ressourcenbedarfs . . . 53
5.3.1 Das Ressourcenmodell . . . 54
5.3.2 Modellierung der Kosten in der Hardware API . . . 56
5.3.3 Abschätzung der Kosten . . . 56
5.4 Vom MPSoC-Modell zur Hardwarekonfigurierung . . . 57
5.4.1 Validierung des MPSoC-Modells . . . 57
5.4.2 Generierung des SPCs . . . 58
5.5 Automatisierung der Werkzeugkette . . . 59

6 Evaluation der entwickelten Werkzeugkette 61
6.1 Durchgängie Konfigurierung einer CAN Anwendung . . . 61
6.1.1 Die Anwendung . . . 61
6.1.2 Implementierung der Threads . . . 62
6.1.3 Durchgängige Konfigurierung von Hard- und Software . . . 62
6.2 Simulation einer UART Applikation . . . 64
6.2.1 Die Anwendung . . . 64
6.2.2 Bewertung der Simulation . . . 66
6.3 Kostenabschätzung von MPSoCs . . . 68

7 Zusammenfassung und Ausblick 71
7.1 Beurteilung von LavA . . . 71
7.2 Zusammenfassung . . . 72
7.3 Ausblick . . . 73

Literaturverzeichnis 77

Abbildungsverzeichnis 79

1 Einleitung

Eingebettete Systeme durchdringen immer mehr Gebiete des täglichen Lebens. Ihre Anwendungen werden dabei immer vielfältiger und komplexer. Die digitale Übertragung von Fernsehsignalen und ihre Ausgabe auf heutigen Fernsehgeräten ist nur ein Beispiel, wie aufwändig die Anwendungen sein können. Um diese Aufgaben bewältigen zu können, benötigen eingebettete Systeme Rechenleistung, die, ähnlich zum Desktop-Bereich, von mehreren Prozessoren übernommen wird bzw. werden muss.

Um eine optimale Ausnutzung der Hardware zu gewährleisten, soll mit dieser Arbeit ein Ansatz verfolgt werden, der erlaubt, dass die benötigte Hardware in der Software selbst deklariert und instanziiert werden kann. Wird beispielsweise ein Gerätetreiber in der Software genutzt, stellt dieser sicher, dass die darunterliegende Hardware ein passendes Exemplar des Gerätes enthält. So wird garantiert, dass Software und Hardware genau aufeinander abgestimmt sind. Ein manueller Umbau der Software, mit eventuell verbundenen Performanzverlusten, wird somit vermieden.

Ermöglicht wird diese Maßschneiderung der Hardware durch den Einsatz von rekonfigurierbarer Logik, den sog. FPGAs, sowie Hardwarebeschreibungssprachen, wie VHDL oder Verilog. Die Hardware wird also textuell beschrieben und lässt sich auf dieser Ebene anpassen. Man kann die Art und Anzahl der Prozessoren, ihre jeweiligen Peripheriegeräte sowie die interne Vernetzung festlegen. Dieses MPSoC wird, zusammen mit der Software, von einem FPGA ausgeführt.

1.1 Das LavA-Projekt

Den Kontext dieser Arbeit bildet das LavA-Projekt (Laufzeitplattform für anwendungsspezifische verteilte Architekturen) [1] der Arbeitsgruppe „Eingebettete Systemsoftware“ des Lehrstuhls 12 der Fakultät für Informatik an der Technischen Universität Dortmund.

Das Ziel des Projekts ist es, sowohl Software als auch Hardware anwendungsgetrieben konfigurieren zu können. Bisher war es möglich, die Software im Hinblick auf ein konkretes Problem maßzuschneidern [2]. Mit diesen Techniken kann die Systemsoftware feingranular konfiguriert werden [3, 4], um den Ansprüchen der jeweiligen Applikation zu genügen. Die Hardware konnte jedoch nicht verändert werden.

Durch die Verwendung von Hardwarebeschreibungssprachen verfällt diese Grenze, sodass erste Arbeiten bereits Hardware konfigurieren [5, 6, 7]. Letztlich soll durch die übergreifende Konfigurierbarkeit eine Plattform entstehen, deren Systemsoftware zusammen mit der Hardware anwendungsspezifisch konfiguriert werden kann.

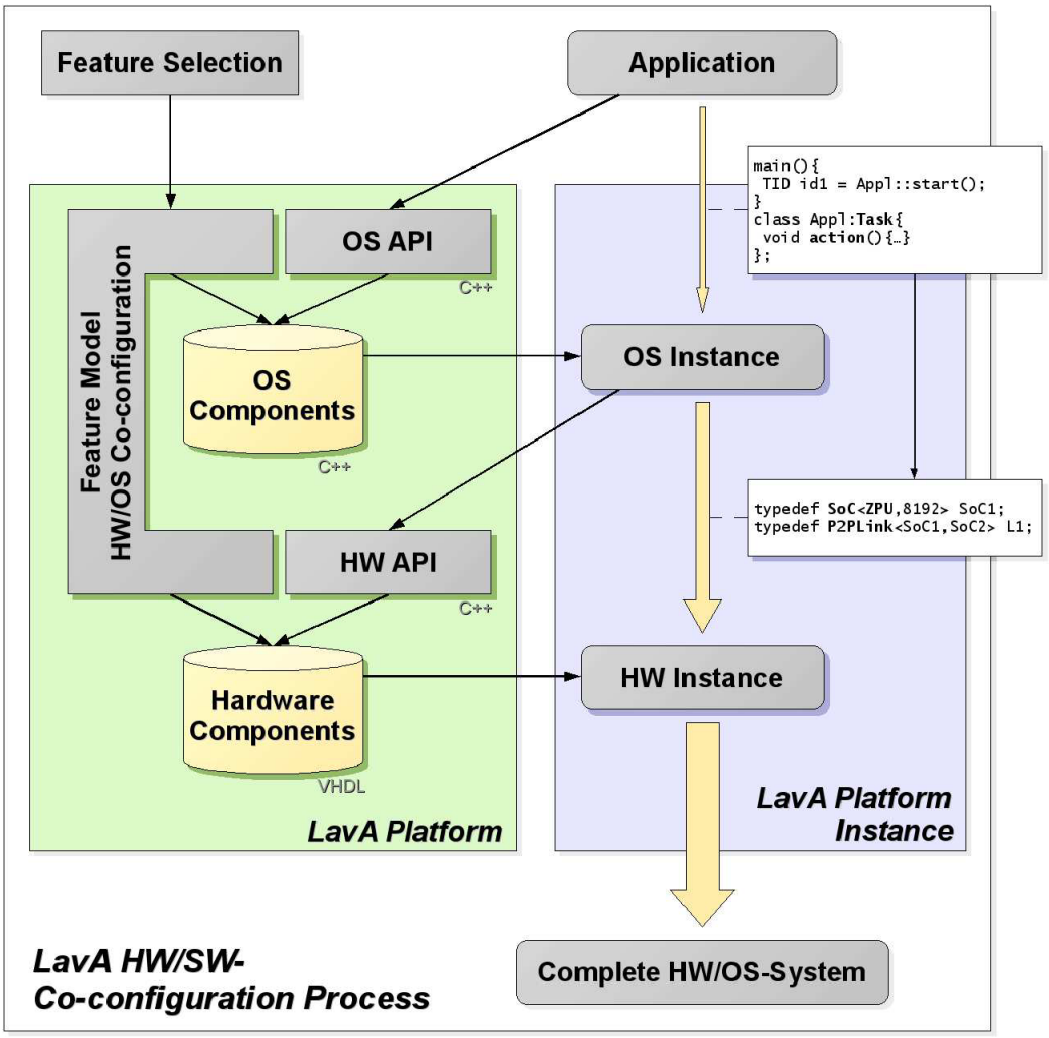

Abbildung 1.1: Schema des LavA-Projekts (Quelle: [1])

Abbildung 1.1 zeigt den schematischen Überblick des LavA-Systems. Das Betriebssystem stellt eine einheitliche Schnittstelle für Anwendungen dar. Eine gegebene Anwendung bestimmt durch ihre Anforderungen, welche Komponenten der Systemsoftware überhaupt benötigt werden. Dies definiert die erste Stufe der Konfigurierung. Die Gerätetreiber der nun vorhandenen Betriebssysteminstanz interagieren mit der Hardware über die sog. Hardware API. Gleichzeitig definiert diese API den Bedarf an Hardware für die Ausführung der Anwendung, was die zweite Stufe der Konfigurierung ausmacht. Die Hardware wird passend zur Anwendung zusammengestellt. Das System, bestehend aus Systemsoftware und Hardware, ist also übergreifend und anwendungsspezifisch maßgeschneidert worden.

Zu den weiteren Bereichen des Projekts zählen die Untersuchung und Entwicklung von konfigurierbaren Hardwareplattformen und ihr Energieverbrauch während der Ausführung, z.B. auf einem FPGA.

1.2 Ziele

Das erste Ziel dieser Arbeit besteht in einer angemessenen Repräsentation von Hardware in Software. Eine Anwendung soll die geforderte Hardware „deklarieren“ können. Gleichzeitig muss diese sog. Hardware API eine elementare Schnittstelle für die Hardware bereitstellen, die sie repräsentiert. Weiterhin muss sie flexibel genug sein, um z.B. mehrere Vorkommen von Peripheriegeräten, auch in unterschiedlichen Variationen, zu ermöglichen.
Den nächster Schritt bildet die automatische Maßschneiderung der Hardwarestruktur durch eine Code-Analyse der Anwendung. Dabei muss auf die Korrektheit der Konfiguration geachtet werden, damit die Hardwareplattform keine internen Fehler, etwa bei der Organisation des Adressraums, aufweist.
Um den Entwickler zu unterstützen, soll eine Werkzeugkette aufgebaut werden, die alle notwendigen Prozesse vereint. Dazu gehört die Suche nach der deklarierten Hardwarestruktur, deren Überprüfung und die Ausführung von Hard- und Software auf einer Zielplattform. Letztere umfassen eine Implementierung auf FPGAs oder Simulationen des Gesamtsystems. Für die Simulation muss eine alternative Variante der Hardware API implementiert werden, zugeschnitten auf die neue Plattform.
Nicht enthalten in dieser Arbeit ist die automatische Aufteilung der Anwendung auf mehrere Rechenkerne. Dies stellt ein Optimierungsproblem dar, welches vorab gelöst werden muss. Es gilt dabei die Kapazitäten der Hardware einzuhalten, jedoch eine akzeptable Ausführungsgeschwindigkeit zu erzielen. Echtzeitfähige Systeme müssen zusätzlich Zeitschranken berücksichtigen.

1.3 Gliederung der Arbeit

Kapitel 2 erläutert die Grundlagen, auf denen diese Diplomarbeit aufbaut. Dazu zählen die modellgetriebene Entwicklung, die Bibliothek Puma, eine konfigurierbare Hardwareplattform und vergleichbare Arbeiten in dem Bereich der Diplomarbeit. Die Maßschneiderung der MPSoCs erfolgt mit einer umfangreichen Werkzeugkette. Deren fundamentale Konzepte stellt Kapitel 3 vor und geht insbesondere auf die entwickelte Hardware API ein. Den Entwurf der Hauptkomponenten, allen voran der Parser, beschreibt Kapitel 4. Die konzeptionellen Arbeiten vertieft Kapitel 5 mit Ausschnitten aus der Realisierung. Kapitel 6 erläutert den Umgang mit der Werkzeugkette. Abschließend reflektiert Kapitel 7 den Gesamtansatz von LavA. Danach wird die Diplomarbeit kurz zusammengefasst, ehe mögliche, zukünftige Arbeiten formuliert werden.

2 Grundlagen

Dieses Kapitel soll die Grundlagen präsentieren, auf denen die späteren Arbeiten aufbauen. Zunächst wird in Kapitel 2.1 die modellgetriebene Softwareentwicklung erläutert, mit deren Hilfe die Hardware API modelliert und generiert wird. Die Hardware API wird später mit einem Programm gesucht, welches auf der C++ Bibliothek Puma basiert. Diese wird in Kapitel 2.2 vorgestellt. Wie aus dem Modell des gewünschten MPSoCs der passende VHDL-Code entsteht, zeigt Kapitel 2.3. Abschließend wird in Kapitel 2.4 eine Übersicht über vergleichbare Arbeiten in diesem Gebiet gegeben.

2.1 Modellgetriebene Softwareentwicklung mit openArchitectureWare

Um eine höhere Abstraktionsebene zu erreichen und damit ein Problem besser beschreiben zu können, nutzt man die Technik der modellgetriebenen Softwareentwicklung (MDSD). Stahl et al. [8] definieren die Thematik als einen „Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen“. Als Vorteile bietet MDSD, durch einheitliche Softwaregenerierung, eine bessere Lesbarkeit und Qualität zu erhalten. Mit lediglich einem zentralen Artefakt, dem Modell, reduziert sich die Zahl möglicher Fehlerquellen. Zum Beispiel wäre der Umbau einer vorhandenen Schnittstelle an nur einem Ort durchzuführen, was die Entwicklung effizienter gestaltet.
Im Mittelpunkt steht also ein formales Modell, welches beispielsweise mit UML [9] oder sog. domänenspezifischen Sprachen beschrieben werden kann. Modelliert wird meist ein Teil des Gesamtsystems, etwa Schnittstellen oder *Frameworks*. Das Modell ist formal, da es einem übergeordneten Metamodell entspricht. Das Metamodell gibt für die Komponenten des Modells Regeln vor, etwa ihre Attribute oder auch Beziehungen untereinander.
Modelle dienen hier nicht ausschließlich der Dokumentation, sondern sollen „lauffähige Software erzeugen“. Dies geschieht durch zwei Möglichkeiten. Zum einen kann mittels Softwaregenerierung, z.B. durch Template-Sprachen, Quelltext entstehen, der dem jeweiligen Modell entspricht. Die Zielsprache ist hierbei lediglich abhängig vom Generator und kann, unabhängig vom Modell, beliebig gewechselt werden. Mit dieser Technik wird die Hardware API aus dem LavA-Metamodell erzeugt (siehe Kapitel 3.2). Zum anderen kann das Modell selbst als Eingabe für einen Interpreter dienen und direkt ausgeführt werden. Das Datenformat der Modelle und Metamodelle ist meist ein XML-Format, sodass die Informationen leicht zugänglich sind. Der Linux-Simulator (siehe Kapitel 4.2) interpretiert ebenfalls das angegebene MPSoC-Modell.

In dieser Arbeit wurde openArchitechtureWare [10] (oAW) in der Version 4.3.1 genutzt. Es handelt sich dabei um ein Plugin für die Entwicklungsumgebung *Eclipse* [11]. Mittlerweile sind die Bestandteile von oAW in dem *Eclipse Modelling Framework* integriert worden. Verwendet wurden die enthaltenen Sprachen für die Überprüfung von Modellen (*Check*), die automatische Generierung von Text (*Xpand*) sowie die Modellierung mit Hilfe des *Ecore*-Metametamodells.

Hauptbestandteil des Metametamodells ist die `EClass`. Eine `EClass` kann über Attribute verfügen, die vom Typ `EAttribute` sind. Die Beziehung zu einer anderen `EClass` wird über eine `EReference` ausgedrückt. Das komplette *Ecore*-Metametamodell, abgebildet durch ein Klassendiagramm, findet sich in Abbildung 2.1.

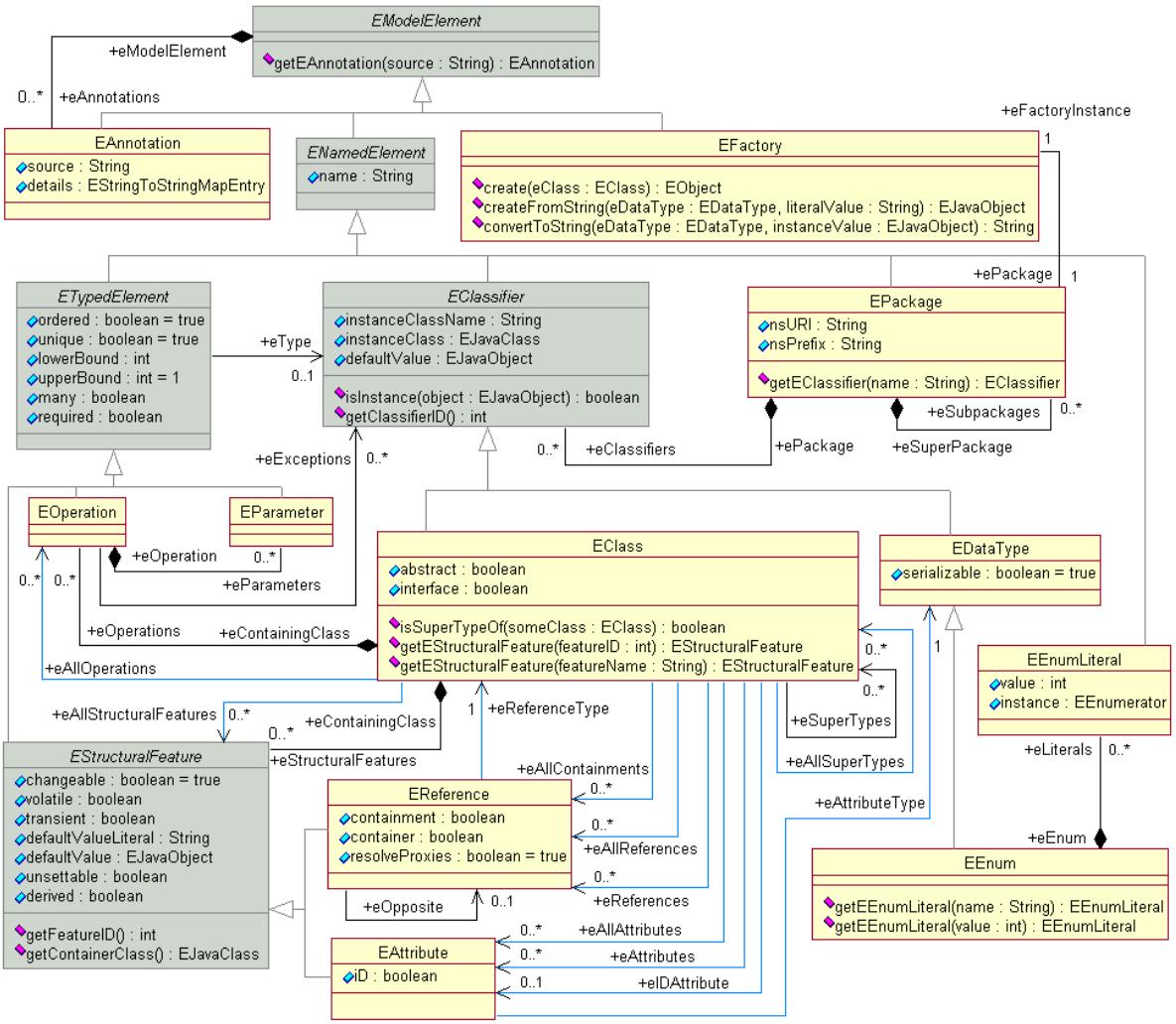

Abbildung 2.1: Das *Ecore*-Metametamodell (Quelle: [11])

2.2 Puma

Puma [12] ist eine Bibliothek, geschrieben in AspectC++, für die Analyse und Manipulation von C, C++ und AspectC++ Quelltext. Die Bibliothek entstand im Rahmen von AspectC++ [2], einer Erweiterung von C++, welche aspektorientierte Programmiertechniken ermöglicht. Veröffentlicht wurde Puma unter der zweiten Version der *GNU General Public License* [13].
Puma umfasst folgende Aspekte:

- Präprozessor für C
- Lexikalische Analyse von Quelltext
- Syntaktische Analyse von Symbolfolgen
- Semantische Analyse von Syntaxbäumen
- Manipulation von Quelltext auf syntaktischer Ebene

Die Hochsprachen C und C++ werden nach den Standards ISO/IEC 9899-1999(E) bzw. ISO/IEC 14882:1998(E) vollständig unterstützt. Weiter enthält Puma zusätzliche Spracherweiterungen von GNU C/C++ und Microsoft Visual C++.

2.3 Die konfigurierbare Hardwareplattform

Eine Möglichkeit, Quelltexte oder sonstige Textdateien statisch zu konfigurieren, ist die Nutzung von XVCL [14]. Bei der *XML-based Variant Configuration Language* wird mit Hilfe sog. *Frames* konfiguriert. Ein *Frame* ist eine XML-Datei, die sowohl Quelltext, als auch XVCL-Befehle enthalten kann. Die Befehle reichen von bedingter und wiederholter Abarbeitung bis zur Erweiterung des aktuellen *Frames* durch den Inhalt anderer *Frames*. Auf diese Weise ermöglicht XVCL Variabilität und Generizität in jeglichen textbasierten Artefakten. Das konkrete Problem, für das XVCL Quelltext generieren soll, wird in dem sog. *Specification Frame*, kurz SPC, angegeben.
In [5] wird XVCL genutzt, um ein konfigurierbares MPSoC aufzubauen, welches auf der Diplomarbeit von Matthias Meier [15] basiert. Der VHDL-Quelltext des Systems ist, wie angesprochen, in *Frames* verteilt. Dabei wird zusätzlich die Abstraktion so gesteigert, dass für die Konfiguration des MPSoCs kein Wissen über die Hardwarebeschreibung oder Hardwarebeschreibungssprachen selbst nötig ist.

Ziel dieser Diplomarbeit ist es also, aus einer gegebenen Anwendung die folgenden Punkte zu extrahieren und im SPC festzulegen:

- Zielarchitektur (FPGA, Taktfrequenz)
- Anzahl der *systems-on-chip* (SoC)
- Kommunikationsstrukturen
- Prozessortyp und Speichergröße pro SoC
- Spezielle Prozessoreigenschaften, wie Multiplizierer
- Unterstützung für Unterbrechungen
- Art und Anzahl der Peripheriegeräte pro SoC
  - *Controller Area Network* (CAN)
  - *Universal Asynchronous Receiver Transmitter* (UART)
  - Unabhängige Zählerregister (Timer)
  - Ein- und Ausgabe-Ports

Neben der bereits aufgeführten Peripherie sind momentan drei verschiedene Prozessortypen in der Hardwareplattform integriert: Die *RISC*-Prozessoren *Plasma MIPS* [16] und *MB-Lite* [17] sowie die *ZPU* [18], die eine Kellermaschine realisiert. Die Prozessoren besitzen unterschiedliche Ausprägungen bzgl. Geschwindigkeit und Flächenverbrauch. Sie liegen jeweils in VHDL vor und sind frei verfügbar. Weiterhin sind Portierungen der *GNU Compiler Collection* [19] vorhanden.

2.4 Related Work

In diesem Kapitel werden zwei Arbeiten vorgestellt, die ebenfalls ein MPSoC durch umfassende Werkzeugketten entwerfen, optimieren und implementieren können. Die Projekte Daedalus [6] und Koski [7] erhalten als Eingabe Quelltexte bzw. UML-Modelle und erstellen ein möglichst optimales MPSoC, welches schließlich auf einem FPGA realisiert wird.

2.4.1 Daedalus

In [6] wird ein Entwurfsprozess vorgestellt, der zugeschnittene MPSoCs für Multimediaanwendungen erstellt. Diese Anwendungen berechnen beispielsweise Filter auf Bildern oder de- bzw. enkodieren diese. Um diese Aufgaben effizienter zu lösen, werden einzelne Teilaufgaben auf mehrere Prozessoren oder Hardwarebeschleuniger aufgeteilt. Abbildung 2.2 zeigt den gesamten Entwicklungsprozess mit Daedalus, dessen Ablauf größtenteils automatisiert erfolgt.

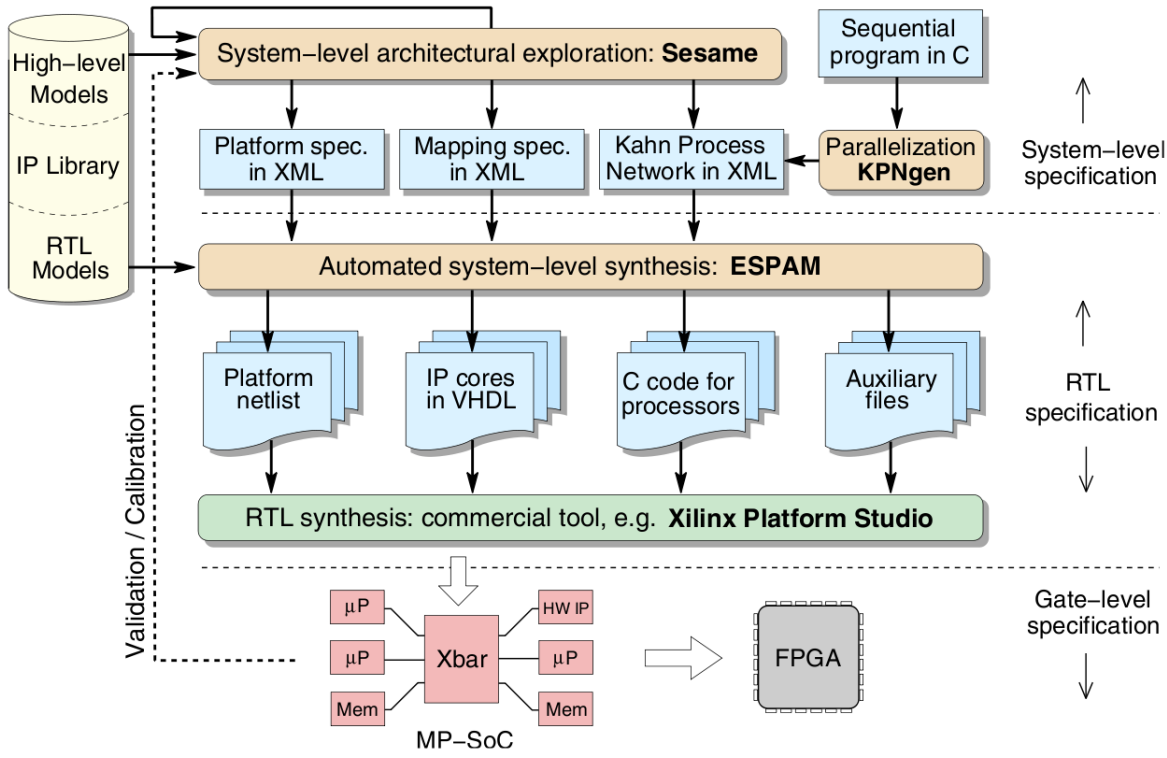

Abbildung 2.2: Übersicht über den Designprozess mit Daedalus (Quelle: [6])

Die Eingabe für Daedalus ist eine sequentielle, in C programmierte Anwendung. Das Programm muss eine gewisse Struktur aufweisen: Auf höchster Ebene dürfen nur sog. statische affine Schleifen verwendet werden. Dies sind Schleifen-Konstrukte, deren Iterationsgrenzen zur Übersetzungszeit bestimmt werden können. Im innersten Schleifenrumpf können dann beliebige Berechnungen ausgeführt werden. Quelltext 2.1 zeigt eine mögliche Eingabe für Daedalus. Parametrisiert werden diese Programme lediglich über globale Konstanten.

```
#define N 128
int main() {
  int array[512], i, j;
  for (i = 0; i < N; i++) {
    for (j = i + 1; j <= 4*N-1; j++) {
      array[j] = compute(array, i, j);
    }
  }
  return 0;
}
```

Quelltext 2.1: Beispielprogramm für Daedalus mit statischen affinen Schleifen

Die erste Aufgabe von Daedalus besteht darin, das sequentielle Programm in ein passendes, paralleles Modell zu überführen. Als Modell haben die Autoren das *Kahn Prozessnetzwerk* [20] gewählt, welches parallele Prozesse und ihre Kommunikation untereinander

abbilden kann. Diese Umwandlung geschieht automatisiert durch das Programm *KPN-gen*. Zusätzlich werden Änderungen am Quelltext vorgenommen, um unterschiedliche, jedoch semantisch äquivalente, Modelle zu generieren. Diese gewonnenen Modelle können weiter variiert werden, um z.B. den Grad der Parallelisierung zu verändern. Man erhält mehrere Modelle als potentielle Lösungen.
Die nun in einem XML-Format vorliegenden Modelle werden von dem Werkzeug *Sesame* untersucht, um die Kandidatenmenge auf die vorab besten Systeme einzugrenzen. Dazu werden für jedes Modell mögliche Hardwarearchitekturen erstellt und anschließend die Software auf die Rechenknoten verteilt. Die Hardwarekomponenten sind dabei relativ abstrakt, um eine schnelle Simulation des Systems zu erreichen. Die gemessene Ausführungszeit der Simulation soll über die Leistung des zugehörigen Systems Aufschluss geben. Über *Tracing* werden zusätzliche Informationen, wie Kommunikation zwischen Prozessen und Rechenlast der Knoten ermittelt.
Der letzte Schritt, realisiert durch *ESPAM*, ist die Implementierung der Systeme und die Bewertung ihrer Ausführung auf einem FPGA. Die von *Sesame* gelieferten Resultate werden in Hard- und Software umgewandelt, um anschließend eine lauffähige Konfiguration für das FPGA zu ergeben. Ausführungszeiten sowie Flächenverbrauch werden gemessen, um zukünftige Ergebnisse von *Sesame* zu verbessern.
Im Vergleich zu dieser Diplomarbeit ist Daedalus in der Art der Eingabe beschränkt. Weiter legt man sich auf den Bereich der Multimediaanwendungen fest. Die möglichen Hardwarekomponenten bestehen daher lediglich aus Prozessoren, Beschleunigern für Berechnungen sowie der Vernetzung dieser Rechenknoten. Diese Arbeit zielt nicht auf einen bestimmten Bereich von Anwendungen und unterstützt ebenso Schnittstellen wie CAN oder UART. Beide Arbeiten haben als Ausgangspunkt ein im Quelltext vorliegendes Programm und verfahren dann automatisiert. Weiter wird bei Daedalus die Architektur abhängig von der Eingabe optimiert. Eine Optimierung des MPSoCs ist ebenfalls für die Werkzeugkette des LavA-Projekts geplant, wird in dieser Diplomarbeit jedoch nicht verwirklicht.

2.4.2 Koski

Der Prozess in [7] beschreibt, ähnlich zu Daedalus, eine Methode, wie man automatisiert aus einer Eingabe, hier UML-Modelle, ein maßgeschneidertes MPSoC erstellt. Die möglichen Anwendungen sind unbegrenzt, obwohl die Autoren Nachteile für Multimediaanwendungen einräumen. In der Abbildung 2.3 wird der Designprozess von Koski veranschaulicht. Auch hierbei ist die Zielplattform ein FPGA, um Prototypen schnell entwickeln und testen zu können. Alle Entwicklungsschritte werden grafisch veranschaulicht, um so die Entwicklung zu vereinfachen.
Zunächst muss der Entwickler funktionale Eigenschaften, wie maximale Laufzeiten innerhalb der Anwendung oder den größtmöglichen Flächenverbrauch der resultierenden Schaltung festlegen. Unter Berücksichtigung dieser Eigenschaften wird dann die Anwendung mit Hilfe von UML-Modellen erstellt. Für die Modellierung wurde ein spezielles UML-Profil entwickelt, was sowohl die Modellierung, als auch die Annotation von Anforderungen und Ergebnissen ermöglicht. Benötigt wird das Modell der Applikation,

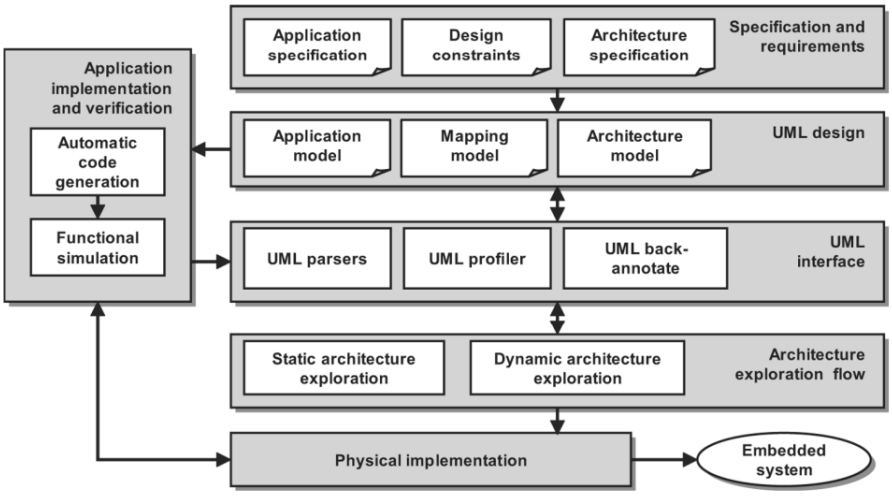

Abbildung 2.3: Übersicht über den Designprozess mit Koski (Quelle: [7])

gegeben durch ein Klassen- und ein Kompositionsdiagramm. Diese Diagramme legen die Prozesse und ihre Beziehungen untereinander fest. Ein Prozess kann dabei wiederum durch ein Kompositionsdiagramm dargestellt werden, um tiefere Strukturen abzubilden, oder aus einem *StateChart* [21] bestehen, um letztlich die Funktionalität auszudrücken. Weitere optionale Modelle sind Kompositionsdiagramme für die Hardwarearchitektur und die Zuordnung der Prozesse auf die Rechenknoten.

Für die bessere interne Verarbeitung werden die vorliegenden Modelle in *Kahn Prozessnetzwerke* umgewandelt. Die erste Suche nach der idealen Kombination aus Hardware und der Zuordnung von Software auf Hardware erfolgt auf einer hohen Abstraktionsebene, damit die nötigen Ergebnisse schnell erzielt werden können. Dazu wird die Architektur inklusive ihrer Software mit einem *Instruction Set Simulator*[1] simuliert. Neben der funktionalen Korrektheit der Anwendung bewertet die Simulation gleichzeitig das aktuelle MPSoC. Dadurch erhält man Kenntnisse über die Verteilung von Rechenlast und Kommunikation, um relativ früh z.B. die Zuordnung der Prozesse auf ihre Rechenkerne zu beeinflussen.

Koski besitzt eine Plattform für die nötigen konfigurierbaren Hardware- und Softwarekomponenten. Die Hardware umfasst Prozessoren, Kommunikationsstrukturen, Hardwarebeschleuniger und diverse Schnittstellen für die Außenanbindung. Die Softwarebibliothek enthält das Betriebssystem *eCos* [22], transparente Routinen für die Interprozesskommunikation sowie eine Abstraktionsschicht der Hardware. Die benötigten Komponenten werden passend zusammengebunden und synthetisiert bzw. kompiliert. Auf Grund der Modellierung mit UML, insbesondere mit *StateCharts*, kann der Quelltext komplett generiert werden. Projektabhängige Algorithmen, etwa eine Prüfsummenberechnung, werden manuell der Softwarebibliothek hinzugefügt, in den *StateCharts* referenziert und bei der Erstellung des Quelltextes eingebunden.

[1] Simuliert Prozessor auf Basis des Instruktionssatzes und erlaubt genaue Messungen der Ausführung

Darauf folgt die zweite Optimierungsphase der noch verbliebenen Kandidaten für das MPSoC auf dem FPGA. Durch die Konfiguration des FPGAs kann das tatsächliche Laufzeitverhalten sowie der Flächenverbrauch gemessen werden. Zusammen mit einer, vom Entwickler spezifizierten, Kostenfunktion werden dann Optimierungsalgorithmen ausgeführt, die die Plattform schrittweise verbessern sollen. Die Resultate der Ausführung und Optimierung fließen zurück in die UML-Modelle. Letztlich ergibt sich so ein auf die Anwendung maßgeschneidertes MPSoC.
Wie auch diese Diplomarbeit, grenzt Koski die Anwendungsbereiche nicht ein. Mit dem Einsatz von UML wird ebenfalls modellgetriebene Entwicklung verwendet. Im starken Kontrast zu Daedalus und dieser Arbeit, besteht bei Koski sogar die Eingabe aus Modellen. Wie bei Daedalus, gilt auch hier, dass eine Optimierung des MPSoCs oder der Zuordnung zwischen Software und Hardware im Rahmen dieser Diplomarbeit nicht stattfindet.

3 Konzeption der Werkzeugkette

In diesem Kapitel wird ein grober Umriss der Werkzeugkette gegeben werden, die die Maßschneiderung von MPSoCs realisieren soll. Die hier getroffenen Entscheidungen beeinflussen maßgeblich die späteren Werkzeuge, die in Kapitel 4 vorgestellt werden. Weiter muss der Entwurf so flexibel sein, dass unterschiedliche Zielarchitekturen, wie z.B. FPGAs oder Simulatoren, unterstützt werden können.

Kapitel 3.1 diskutiert die Möglichkeiten, den Bedarf von Hardware in Software umzusetzen. Dazu werden die jeweiligen Konsequenzen für die weitere Konfigurierung aufgezeigt. Die letztlich gefundene Lösung wird in Kapitel 3.2 genauer erläutert. Abschließend gibt Kapitel 3.3 einen Überblick der zu realisierenden Werkzeugkette.

3.1 Repräsentation von Hardware in Software

Anwendungen für eingebettete Systeme werden meist in C oder C++ implementiert. Es gilt demnach, ein Sprachmittel zu finden, um Hardware im Quelltext zu deklarieren. Grundvoraussetzung ist, dass das Konstrukt statisch, zur Übersetzungszeit, wirkt. Dadurch kann die Analyse mit einem Parser (siehe Kapitel 4.1) die gewünschte Hardware finden. Weiterhin sollten folgende Anforderungen erfüllt werden können:

- Unterschiedliche Variationen gleicher Bausteine: Es muss möglich sein mehrere Geräte gleichen Typs mit jeweils unterschiedlicher Konfiguration in das MPSoC zu integrieren.
- Kommunikation mit der Peripherie: Die Kommunikation zwischen Prozessor und Peripherie erfolgt über Speicherzugriffe. Die gefundene Lösung muss hierfür eine elementare Schnittstelle bereitstellen.
- Aufteilung des Adressraums: Um Probleme bei der Kommunikation mit der Peripherie zu vermeiden, muss der Adressraum so organisiert werden, dass die Bereiche der einzelnen Geräteinstanzen sich nicht überschneiden.
- Zuweisung der Unterbrechungsnummer: Peripheriegeräte können den Prozessor über Unterbrechungen benachrichtigen. Die Vergabe der Unterbrechungsnummern muss daher geregelt werden.
- Variable Implementierung: Die Unterstützung mehrerer Zielarchitekturen soll durch einfache Änderungen möglich sein.

Die o.g. Grundvoraussetzung wird durch das Sprachmittel der C++ *Templates* erfüllt. Daher beruhen die folgenden Lösungsansätze auf diesem Konstrukt, welches zunächst, im Kontext der Diplomarbeit, erläutert wird.

3.1.1 C++ Templates

Templates sind ein mächtiges Sprachkonstrukt von C++. Sie bieten Variabilität, um Algorithmen einmalig zu definieren, welche dann durch den Übersetzer passend zum Zieltyp generiert werden. Die Wiederverwendbarkeit und Wartbarbarkeit wird, durch eine einmalige Implementierung, verbessert. Als Beispiel dient die in Quelltext 3.1 dargestellte Berechnung des Maximums für jene Datentypen, auf denen der >-Operator definiert ist.

```
template<typename T>
T max(T x, T y) {
  return x > y ? x : y;
}

void test() {
  int a = 3, b = 7, c;
  c = max<int>(a, b);

  float pi = 3.1415, e = 2.7182, f;
  f = max<float>(pi, e);
}
```

Quelltext 3.1: Berechnung von Maxima durch *Templates*

Wegen dieser Vorteile wurden *Templates* intensiv eingesetzt, um der Sprache C++ eine Bibliothek konfigurierbarer Algorithmen und Datenstrukturen hinzuzufügen – die *Standard Template Library*, kurz STL. Weitere Arbeiten mit *Templates* sind in [23, 24] detailliert beschrieben.
Templates entsprechen also den o.g. Anforderungen, da sie zum Zeitpunkt der Übersetzung gehandhabt werden können und verschiedene Variationen ermöglichen. Ein Beispiel – mit mehr Bezug zum Thema MPSoCs – ist der `UART` mit einer konfigurierbaren Baudrate. Realisiert durch *Template*-Klassen, zeigt Quelltext 3.2 eine sehr einfache Variante für eine mögliche Abbildung des UARTs in C++. Die tatsächlichen Typen der Objekte `uart1` und `uart2` sind dem Übersetzer bekannt, der dazu die Rechnung im *Template*-Parameter von `UART<3*19200>` durchführen muss.

```
template<int Baud>
class UART {};

UART<115200>  uart1;
UART<3*19200> uart2;
```

Quelltext 3.2: UART als *Template*-Klasse

3.1.2 Aufruf von statischen Methoden

Die erste mögliche Lösung, geforderte Hardware in Software zu deklarieren, sind Aufrufe von statischen Methoden. Die Hardware wird durch *Template*-Klassen definiert, über deren Parameter sie konfiguriert werden kann. Die Schnittstelle für den Zugriff benötigt zur Unterscheidung die Instanznummer des UARTs. Eine Veranschaulichung bietet Quelltext 3.3. Der implementierte Speicherzugriff muss zunächst die konkrete Adresse auflösen. Die dafür benötigten Variablen sind alle bei der Übersetzung bekannt, sodass der Übersetzer diese Adresse direkt berechnen kann.

```
template<int Baud>
struct UART {
  enum { AddressSpacePerInstance = 0x1000 };
  enum { AddressSpaceBase = 0x80201000 };

  template<int instanceNr>
  static unsigned int address() {
    return AddressSpaceBase + AddressSpacePerInstance * instanceNr;
  }

  template<int instanceNr>
  static void write(unsigned int offset, int data) {
    int *p = (int *) (address<instanceNr>() + offset);
    *p = data;
  }
};

void ignore() {
  // Niemals aufgerufene Funktion
  UART<57600>::write<1>(0x4, 0xFF);
}

int main() {
  // Zwei Schreibzugriffe auf UART0
  UART<57600>::write<0>(0x4, 0xAB);
  UART<38400>::write<0>(0x4, 0xCD);
}
```

Quelltext 3.3: Hardwareinstanziierung durch statische Methoden

Die Suche nach der benötigten Hardware muss also bei allen Aufrufen statischer Methoden den vorangestellten Typ untersuchen. Falls dieser Typ eine Hardwareklasse ist, wird eine Instanz, mit entsprechender Konfiguration, dem MPSoC hinzugefügt.
Leider ergeben sich hier noch Schwierigkeiten. In Zeile 20 ist ein Zugriff auf den `UART1` aufgeführt. Die den Zugriff umgebende Funktion `ignore` wird jedoch nicht genutzt. Letztlich findet die Interaktion mit dem UART also nicht statt. Demnach muss der gesamte Aufrufgraph der Anwendung aufgebaut werden, um solche falschen Treffer auszuschließen. Dies kann bei größeren Anwendungen sehr aufwendig werden, da potentielle Aufrufe aus allen Übersetzungseinheiten möglich sind. Zusätzlich erschweren Zeiger auf Funktionen dieses Problem.

Ein weiterer Nachteil der Lösung ist in den Zeilen 25f aufgeführt. Dort ergibt sich ein Konflikt, der sowohl die Instanznummer, als auch die Baudrate betrifft. Hier ist unklar, ob die zwei Aufrufe dem selben UART gelten und damit eine Baudrate falsch ist, oder ob wirklich zwei Geräte instanziiert werden sollen und so eine Instanznummer fehlerhaft ist. Eine (halb-)automatische Korrektur ist daher nicht möglich.
Problematisch ist dieser Ansatz ebenfalls bei Hardware, die keine Schnittstelle besitzt. Zum Beispiel muss der Prozessor nicht notwendigerweise eine derartige Funktion besitzen. Die Möglichkeit, eine Methode aufzurufen, entfällt. Abhilfe schaffen hier nur unnötige, leere Methoden, die dann genutzt werden müssen.
Durch den Aufruf der statischen Methoden wird durch C++ weder ein Objekt angelegt noch Speicher reserviert. Im Widerspruch dazu wird allerdings eine Hardwarekomponente instanziiert. Dies stellt einen Bruch mit der eigentlichen Semantik und die nächste Schwierigkeit dar.

3.1.3 Typdefinitionen

Der nächste Ansatz stellt eine leichte Veränderung zum vorherigen dar. Nun sind Typdefinitionen das entscheidende Kriterium, ob Hardware genutzt wird oder nicht. Der Aufrufgraph wird also nicht benötigt. Ebenso überflüssig sind die leeren Methoden, die bisher zwingend genutzt werden mussten. Die Definition der *Template*-Klassen kann übernommen werden, was Quelltext 3.4 zeigt.

```
template<int Baud>
struct UART {
  /* siehe Quelltext 3.3 */
};

typedef UART<57600> UART0;
typedef UART<38400> UART1;

int main() {
        UART0::write<0>(0x4, 0xAB);
        UART1::write<1>(0x4, 0xCD);
}
```

Quelltext 3.4: Hardwareinstanziierung durch Typdefinition

Positiv zu bewerten ist die einfachere Suche nach der Verwendung von Hardware, im Vergleich zum vorigen Kapitel. Die Typdefinitionen können dezentral in der Anwendung verteilt sein. Ein Gerätetreiber kann daher die Definition beherbergen und sorgt damit für die Hardware in exakt der Konfiguration, wie er sie benötigt. Allerdings ist man immer noch an statische Methoden gebunden, da primär keine Objekte angelegt werden. Werden dennoch Objekte mit Hilfe der Typdefinitionen allokiert, so entspricht dies dem folgenden Ansatz.
Durch die geringfügige Änderung der beiden bisherigen Lösungen, gelten weiterhin die bereits präsentierten Nachteile und Schwierigkeiten.

3.1.4 Globale Objekte

Diese Strategie verwendet neben definierten Typen konkrete Objekte. Hardware wird dadurch repräsentiert, dass ein globales Objekt existiert, dessen Typ eine Hardwareklasse ist. Diese Objekte können nicht-statische Attribute und Methoden besitzen. Daher ist es möglich, dass die Speicheradressen der Peripherie in den Objekten abgelegt werden können und nicht mehr bei jedem Aufruf mit Hilfe der Instanznummer ermittelt werden müssen. Die wichtigsten Unterschiede liegen in den Zeilen 3 und 19–29 des folgenden Quelltextes 3.5.

```
template<int Baud>
class UART {
  unsigned int address;
public:
  enum { AddressSpacePerInstance = 0x1000 };
  enum { AddressSpaceBase = 0x80201000 };

  template<int instanceNr>
  void instance() {
    address = AddressSpaceBase + AddressSpacePerInstance * instanceNr;
  }

  void write(unsigned int offset, int data) {
    int *p = (int *) (address + offset);
    *p = data;
  }
};

UART<57600> uart0;
UART<38400> uart1;

int main() {
  // Initialisierung der Basisadressen
  uart0.instance<0>();
  uart1.instance<1>();

  // Schreibzugriffe
  uart0.write(0x4, 0xAB);
  uart1.write(0x4, 0xCD);
}
```

Quelltext 3.5: Hardwareinstanziierung durch Typdefinition

Mit dieser Lösung müssen also globale Objekte bestimmter Klassen gefunden werden. Globale Objekte sind hier wie folgt definiert (vgl. dazu Quelltext 3.8):

1. Nicht-statische globale Objekte
2. Statische Attribute von Klassen
3. Rückgabewerte von Singleton-Methoden

Ihr Einsatz ist also sehr flexibel. Eine zusätzliche Hardwareinstanz mit gleicher Konfiguration erhält man durch die Deklaration eines weiteren Objekts. An die Instanziierung von Hardware ist gleichzeitig eine Allokation von Speicherplatz für mindestens die Basisadresse gekoppelt, sodass der Semantik von C++ weitgehend entsprochen wird. Leider wird durch die explizite Speicherung der Basisadresse mehr Programmcode generiert, denn bei jedem Speicherzugriff wird die Adresse zunächst aus dem Speicher geladen.

Der größte Vorteil von diesem Ansatz liegt in der Einbindung der Hardwareklassen in die Architektur der Anwendung. Gerätetreiber können Hardware sowohl als Attribute beherbergen als auch von den jeweiligen Klassen erben, um dann selbst durch ein Objekt instanziiert zu werden. Die möglichen gültigen Konstrukte durch Vererbung und Komposition sind an dieser Stelle unbeschränkt und werden in Kapitel 3.2.2 veranschaulicht.

Als ein Nachteil ist der zusätzliche Aufwand durch die explizite Angabe der Instanznummer mit Hilfe der Methode `instance` aufzuführen. Diese Lösung ist jedoch weniger fehleranfällig als die beiden ersten, denn ein Problem kann lediglich auftreten, wenn mehrere unterschiedliche Instanznummern dem selben Objekt zugeordnet werden. Eine Konfiguration der Instanznummer über den Konstruktor der Klasse scheidet aus, da die Reihenfolge der Konstruktoraufrufe nicht festgelegt werden kann. In Kapitel 3.2.1.3 wird die Bedeutung der Instanznummer vertieft.

Auf Grund der hohen Flexibilität, der Einhaltung der Semantik von C++ und der guten Integrierbarkeit durch Vererbung und Komposition, wurde diese Strategie ausgewählt, um die im folgenden Kapitel beschriebene Hardware API zu implementieren.

3.2 Die Hardware API

Die Hardware API muss eine einfache Schnittstelle für die Interaktion mit der Hardware bereitstellen und bildet daher die logisch unterste Ebene der Software. Um ihre Umsetzung flexibel und leicht erweiterbar zu gestalten, wird sie modellgetrieben entwickelt, was Kapitel 3.2.1 vorgestellt wird. Gleichzeitig ist sie Ankerpunkt für die Konfigurierung des anwendungsspezifischen MPSoCs. Die möglichen Arten der Instanziierung präsentiert Kapitel 3.2.2. Zuletzt zeigt Kapitel 3.2.3, wie die Hardware API Alternativen bieten kann, um z.B. eine Simulation zu ermöglichen.

3.2.1 Modellgetriebene Entwicklung der Hardware API

Die modellgetriebene Entwicklung wurde in Kapitel 2.1 beschrieben. Mit Hilfe dieser Methodik soll hier die Hardware API entwickelt werden. Dazu wird zuerst ein Metamodell benötigt, welches die Struktur der MPSoCs beschreibt. Anschließend wird das Metamodell durch verschiedene Regeln validiert, bevor die eigentliche Generierung Quelltext, in Form von C++ Headerdateien, erzeugen kann.

3.2.1.1 Das Metamodell der Hardware API

Das Metamodell muss alle Komponenten abbilden, um die Hardwareplattform von LavA voll auszuschöpfen. Abbildung 3.1 skizziert das Metamodell in einer ersten Übersicht.

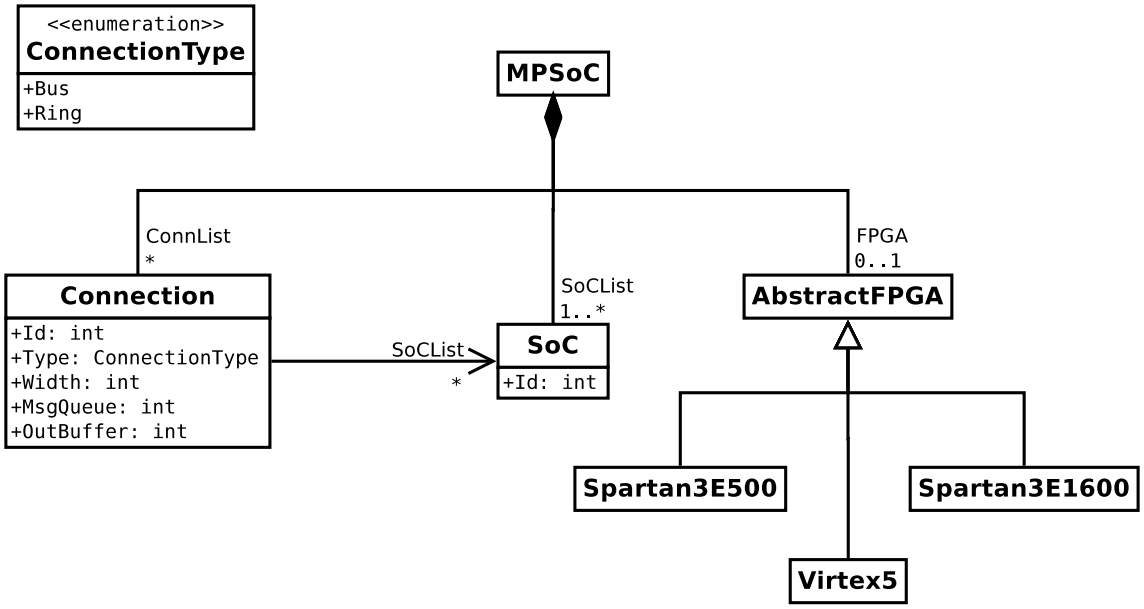

Abbildung 3.1: Das Metamodell der Hardware API

Auf oberster Ebene besteht das MPSoC aus einem oder mehreren SoCs, die durch `Connections` verbunden werden. Falls nur ein einzelnes SoC instanziiert werden soll, werden keine `Connections` benötigt. Die Kommunikationsstrukturen sind sehr konfigurierbar: Neben dem grundsätzlichen Aufbau als Ring oder Bus kann die Datenbreite und die Größe der Puffer auf Sender- und Empfängerseite eingestellt werden. Die SoCs und `Connections` erhalten, durch ihre Identifikationsnummern, jeweils eindeutige Bezeichnungen. Als Zielplattform kann ein FPGA spezifiziert werden. Zur Auswahl stehen momentan die Modelle E500 und E1600 der Baureihe Spartan-3 sowie das Virtex-5 LX110T. Alle drei FPGAs sind von der Firma Xilinx [25] entwickelt worden und am Lehrstuhl verfügbar. Weitere Modelle, auch anderer Hersteller, können leicht hinzugefügt werden. Die Struktur der SoCs präsentiert Abbildung 3.2 – zur Vereinfachung wurden in dieser Übersicht technische Details der modellgetriebenen Entwicklung ausgelassen, um bei gegebener Stelle aufgegriffen zu werden. Im Kern enthalten SoCs eine CPU und eine variable Anzahl an Peripherie.
Die meisten Geräte lassen sich der jeweiligen Aufgabe anpassen. Zum Beispiel kann das Zählerregister des Timers, je nach Anforderung, 32Bit oder 64Bit breit sein. Der Arbeitsspeicher aller Prozessoren kann ebenfalls eingestellt werden. Die Granularität wird durch das verwendete FPGA bestimmt, da sich der Speicher aus den internen RAM-Bausteinen zusammensetzt, welche 2KB (Spartan-3) oder 4KB (Virtex-5) groß sind. Die Kommunikationscontroller IPC erben ihre Eigenschaften von der angeschlossenen Verbindung. Die entsprechend gültigen Werte aller Konfigurationspunkte werden durch die Hardwareplattform aus Kapitel 2.3 definiert.

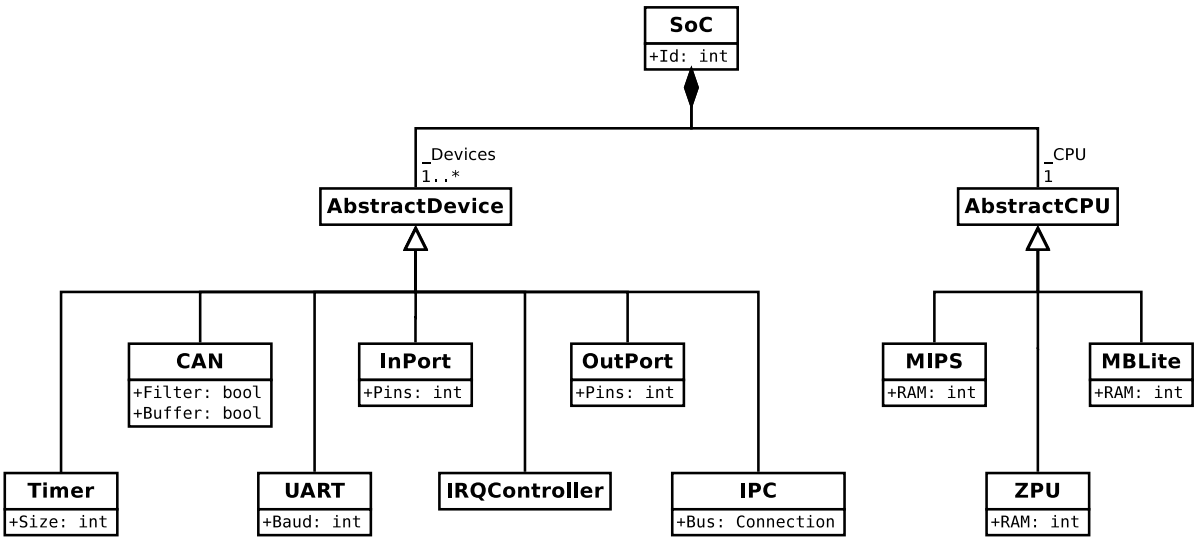

Abbildung 3.2: Die Struktur der SoCs

3.2.1.2 Validierung des Metamodells

Da aus dem Metamodell C++ Klassen generiert werden, müssen zahlreiche Überprüfungen durchgeführt werden, die die Korrektheit des generierten Programmcodes sicherstellen. Dies geschieht in oAW mit der Sprache *Check*. Mögliche Fehler können die Klassennamen der Hardware API betreffen. Diese müssen im Sinne von C++ Klassennamen syntaktisch korrekt sein. Eine von mehreren Abfragen hierzu zeigt Quelltext 3.6.

```
context EClass ERROR
  "Name ungueltig (" + name + ")" :
  name.matches("[_a-zA-Z]\\w*");
```

Quelltext 3.6: Überprüfung der Klassennamen durch einen regulären Ausdruck

Die Regel setzt zunächst den Bereich, auf den sie angewendet wird. Sie betrifft alle Elemente vom Typ `EClass`. Das Schlüsselwort `ERROR` erzwingt einen Abbruch der Generierung, wenn die Bedingung der letzten Zeile scheitert. Die Klassennamen werden mit einem regulären Ausdruck überprüft, um u.a. führende Ziffern auszuschließen. Die Ausgabe liefert dann die entsprechende Fehlermeldung, wie sie in Zeile 2 definiert wurde. Die Variable `name` wird durch den konkreten Namen der `EClass` ersetzt, bei der die Regel fehlschlug. Möchte man nur eine Warnmeldung erhalten, ohne den Prozess zu stoppen, so kann `ERROR` durch `WARNING` ersetzt werden.

3.2.1.3 Konventionen und Einschränkungen bei der Modellierung

Für die korrekte Analyse der Hardware API durch den in Kapitel 4.1 vorgestellten Parser müssen einige Konventionen eingehalten werden. Zum einen benötigt der Parser bei einer gefundenen Instanziierung Informationen über deren Typ. Ein Prozessor wird dabei anders gehandhabt als ein zusätzlicher UART. Zum anderen wird das durch den Parser

erzeugte MPSoC-Modell mit oAW verarbeitet. Hierzu müssen Angaben wie Unterbrechungsnummern korrekt annotiert sein, damit die erzeugte Hardwarebeschreibung und Software zueinander passen.
Der Parser untersucht die Klasse jedes gefundenen globalen Objekts auf Zugehörigkeit zur Hardware API. Die Zugehörigkeit bestimmt der C++ Namensraum, in dem die Klasse definiert wurde. Momentan wird die Hardware API in dem Namensraum `hw::api` platziert. Diese Einstellung wird im LavA-Package des Metamodells unter dem Eintrag `Ns URI` gespeichert.
Das *Ecore*-Metametamodell stellt für die Modellierung einer `EClass` die Möglichkeiten Attribute über `EAttribute` und Referenzen auf andere `EClasses` über `EReference` zu setzen (vgl. Abbildung 2.1). Elemente beiden Typs werden durch die Softwaregenerierung zu *Template*-Parametern. Allerdings benötigt man Ausnahmen, sodass spezifizierte Attribute innerhalb der *Template*-Klassen platziert werden können, z.B. mit Hilfe von *Enumerations*. Um dies zu erreichen, wurde die Namenskonvention eingeführt, dass Attribute und Referenzen, deren Namen mit einem Unterstrich beginnen, nicht als *Template*-Parameter verarbeitet werden.
Weiterhin gibt es Einschränkungen bei den möglichen Typen der *Template*-Parameter. C++ verbietet hier Datentypen für Gleitkommazahlen und Zeichenketten. Bislang stellt dies jedoch kein Hindernis dar, denn die meisten Konfigurationspunkte der Hardwareplattform werden durch ganzzahlige Datentypen repräsentiert.
Nachfolgend werden weitere Konventionen aufgelistet. Diese betreffen die Klassifizierung der unterschiedlichen Hardwareklassen sowie die Berechnung der gerätespezifischen Basisadressen. Zuletzt wird das Attribut vorgestellt, mit dem Geräte signalisieren, dass sie Unterbrechungen verwenden können. Alle Konventionen können, dank der integrierten Sprache *Check*, statisch überprüft werden.

Klassifizierung der Hardware API

Ein SoC muss genau einen Prozessor und kann Peripherie enthalten. Damit der Parser diese Regel überprüfen kann, müssen die einzelnen Instanzen für ihn klassifizierbar sein. Hierzu sind ein bis zwei Angaben im Metamodell notwendig. Es wurden abstrakte Oberklassen definiert, von denen die einzelnen `EClasses` erben müssen. Dazu wird ein entsprechender Eintrag in dem Feld `ESuperTypes` gesetzt. Nebenbei erhalten die Geräte, über die Oberklasse `AbstractDevice`, ihre Speicheradresse und -zugriffsmethoden.
In Tabelle 3.1 sind die notwendigen Angaben, getrennt nach Prozessor, Peripherie und FPGA, zusammengefasst.

Typ	Oberklasse	Wert von `_ClassType`
Prozessor	`AbstractCPU`	„Processor“
Peripherie	`AbstractDevice`	„InternalDevice“ / „ExternalDevice“
FPGA	`AbstractFPGA`	

Tabelle 3.1: Klassifizierung der Hardware API

Eine zweite Einteilung wird über das Attribut `_ClassType` vom Typ `ClassType` getroffen. Die Geräte teilen sich auf in „InternalDevice“ und „ExternalDevice“. Dies gibt Aufschluss darüber, ob das Gerät nur intern genutzt wird, wie der `Timer`, oder auch Anschlusspins des FPGAs benötigt, wie der `OutPort`. Falls es diese externe Anbindung erfordert, muss zusätzlich das Attribut `_UCF` deklariert werden, welches die genaue Position und Kombination der Anschlusspins enthalten kann. Die Hardwareplattform stellt, abhängig vom FPGA, mehrere Möglichkeiten zur Auswahl. Der Entwickler spezifiziert den konkreten Anschluss mit dem Parameter der *Template*-Methode `instance`.

Berechnung der Geräteadressen

Für ein System gilt, dass mehrere Geräte vom selben Typ im selben SoC enthalten sein können. Daher muss die Softwareschnittstelle mit mehreren Instanzen umgehen können. Die Peripherie besitzt hierzu das Attribut `address`, welches durch die *Template*-Methode `instance` einmalig berechnet wird. Das Schema der Rechnung ist in Zeile 10 von Quelltext 3.5 zu sehen. Die dort angegebenen Größen `AddressSpaceBase` und `AddressSpacePerInstance` werden durch die Attribute `_Base` respektive `_Size` modelliert und sind bei Geräten zwingend erforderlich.
Komplettiert wird die Berechnung über die zusätzliche Addition von `CPU::_IO_Base`. Der Wert `_IO_Base` spezifiziert in jeder Prozessorklasse den Start des Speicherbereichs für die Geräte. Eine Typdefinition des verwendeten Prozessors auf den Namen `CPU` muss existieren, damit die o.g. Rahmenbedingungen eingehalten werden.
Eine Übersicht der notwendigen Attribute gibt Tabelle 3.2

Typ	Attributname	Beispielwert
Prozessor	`_IO_Base`	0x80000000
Peripherie	`_Base`	0x201000
	`_Size`	4096

Tabelle 3.2: Notwendige Angaben zur Berechnung der Geräteadressen

Geräte mit unterstützten Unterbrechungen

Falls das Gerät Unterbrechungen unterstützt, benötigt es das Attribut `_IRQ`. Der Wert ist, wie bei `_UCF` auch, unerheblich, denn er wird während der Analyse vom Parser gesetzt.

3.2.1.4 Automatische Generierung der Hardwareklassen

Damit eine LavA-Anwendung die Hardware API nutzen kann, müssen zuvor die Hardwareklassen automatisiert generiert werden. Für die Übersetzung eines Modells in Text wurde die *Template*-Sprache *Xpand* entwickelt. Die in der Anwendung instanziierte Klassenbibliothek bestimmt die Konfiguration des MPSoCs. Daher müssen alle Konfigurationspunkte nun durch *Template*-Parameter abgebildet werden können. Dies betrifft die Prozessoren, die gesamte Peripherie sowie die Klassen `SoC` und `Connection`. Quelltext 3.7 zeigt einen Teil der Generierung.

```
«DEFINE templateParameters FOR EClass»
  «IF !eStructuralFeatures.reject(e|e.name.startsWith("_")).isEmpty»
    template <
    «FOREACH eStructuralFeatures.reject(e|e.name.startsWith("_"))
              AS feat SEPARATOR ",\n"»
      «IF feat.metaType == EAttribute»
        «EXPAND printSingleParameter FOR (EAttribute)feat»
      «ELSE»
        «EXPAND printSingleParameter FOR (EReference)feat»
      «ENDIF»
    «ENDFOREACH»
    >
  «ENDIF»
«ENDDEFINE»

«DEFINE printSingleParameter FOR EAttribute»
  «eType.instanceTypeName»  «name»
    «defaultValueLiteral != null ? "= " + defaultValueLiteral : ""»
«ENDDEFINE»

«DEFINE printSingleParameter FOR EReference»
  typename  «name»  = NullType
«ENDDEFINE»
```

Quelltext 3.7: Generierung der *Template*-Parameter

Dieser Ausschnitt erzeugt im Quelltext der Hardware API die *Template*-Parameter. Anweisungen der Sprache *Xpand* sind durch französische Anführungszeichen markiert und werden zur Laufzeit ausgewertet. Funktionen stehen zwischen den Schlüsselwörtern `DEFINE` und `ENDDEFINE`. Eine Funktion bezieht sich jeweils auf bestimmte Typen, welche hinter `FOR` spezifiziert werden. Weitere Befehle, wie bedingte Ausführung oder Schleifen, sind intuitiv gekennzeichnet. Für jede Hardware API Klasse wird die Funktion `templateParameters` aufgerufen, um vor dem Klassenrumpf die eventuell vorhandenen Parameter zu expandieren.

Ob eine Klasse überhaupt *Template*-Parameter benötigt, muss zuerst festgestellt werden, denn eine *Template*-Definition ohne Parameter bildet einen Syntaxfehler. Die Abfrage in Zeile 2 prüft, ob die `EClass` mindestens ein Feature, also ein Attribut oder eine Referenz, besitzt. Nicht beachtet werden hier die Features, deren Namen unter die vorher festgelegte Namenskonvention fallen.

Gemäß dem Fall, dass Parameter vorhanden sind, wird zunächst der Text „template“ gedruckt (Zeile 3), mit dem jedes *Template* beginnt. Anschließend werden alle Features abgearbeitet und individuell übersetzt.

Für ein `EAttribute` springt die Generierung zur Zeile 16. Ein Wertparameter besteht aus dem Typ, meist einer *Enumeration* oder arithmetischer Typ, dem Namen und, falls vorhanden, dem Standardwert. Der Standardwert kann im Metamodell durch das Feld `defaultValueLiteral` angegeben werden.

Mit `EReferences` wird ab Zeile 21 fortgefahren. Referenzen werden durch einen Typparameter des *Templates* abgebildet. Daher wird das Schlüsselwort „typename“ gefolgt vom Namen der Referenz und dem Standardwert erzeugt. Der Standardwert ist hierbei immer die vollkommen leere Klasse `NullType`, die der Hardware API beiliegt.
Als weiteren Aspekt ermöglicht die Generierung, die Speicheradressen der Peripherie zu spezifizieren. Diese werden in einer `EClass` durch Attribute vom Typ `MemAddress` hinzugefügt und enthalten die entsprechenden Offsets. Daraus werden *Enumerations* generiert, die in der Software genutzt werden können.

3.2.2 Instanziierung der Hardware API

In diesem Kapitel werden die verschiedenen Arten der Instanziierung erläutert. Die in Kapitel 3.1.4 geschilderte Strategie bietet dazu einige Alternativen, die hier veranschaulicht werden sollen.

```
// Objekt im globalen Namensraum
hw::api::UART<> globalUART;                          // Eine Instanz

// Mehrfache Instanziierung mit einem Array
hw::api::UART<> threeUARTs[3];                       // Drei Instanzen

// Instanziierung nach Vererbung
class DerivedUART : public hw::api::UART<> {};
DerivedUART derivedUART;                             // Eine Instanz

// Instanziierung nach Komposition
class KompUART {
public:
  hw::api::UART<> uart1, uart2;
};
KompUART kompUART;                                   // Zwei Instanzen

// Instanziierung in Singleton-Methode
class Singleton : public hw::api::UART<> {
public:
  static Singleton& Inst() {                         // Eine Instanz
    static Singleton theSingleton;
    return theSingleton;
  }
};
```

Quelltext 3.8: Instanziierung der Hardware API

Die Peripherie muss direkt über die Methode `instance` initialisiert werden, damit die Speicheradresse, vor dem ersten Zugriff, korrekt eingestellt ist. Dazu muss die Methode auf dem Objekt der Hardware API aufgerufen werden, ansonsten kann der Parser den Aufruf dem jeweiligen Objekt nicht zuordnen. Daher wurden die Attribute `uart1` und `uart2` in der Klasse `KompUART` als öffentlich deklariert. Die Initialisierung zu den obigen Beispielen zeigt Quelltext 3.9.

```
void init() {
  globalUART.instance<0>();
  threeUARTs[0].instance<1>();
  threeUARTs[1].instance<2>();
  threeUARTs[2].instance<3>();
  derivedUART.instance<4>();
  kompUART.uart1.instance<5>();
  kompUART.uart2.instance<6>();
  Singleton::Inst().instance<7>();
}
```

Quelltext 3.9: Initialisierung der Hardware API

3.2.3 Unterschiedliche Zielarchitekturen

Damit das MPSoC inklusive der Anwendung zeitnah getestet werden kann, muss die Hardware API, neben der Implementierung auf einem FPGA, unterschiedliche Zielarchitekturen unterstützen. Die in Kapitel 4.2 vorgestellte Alternative ist ein Simulator, der auf einem x86-Prozessor mit Linux ausgeführt werden kann. In diesem Kapitel wird die Methode vorgestellt, wie die Hardware API kontextabhängige Implementierungen bieten kann.

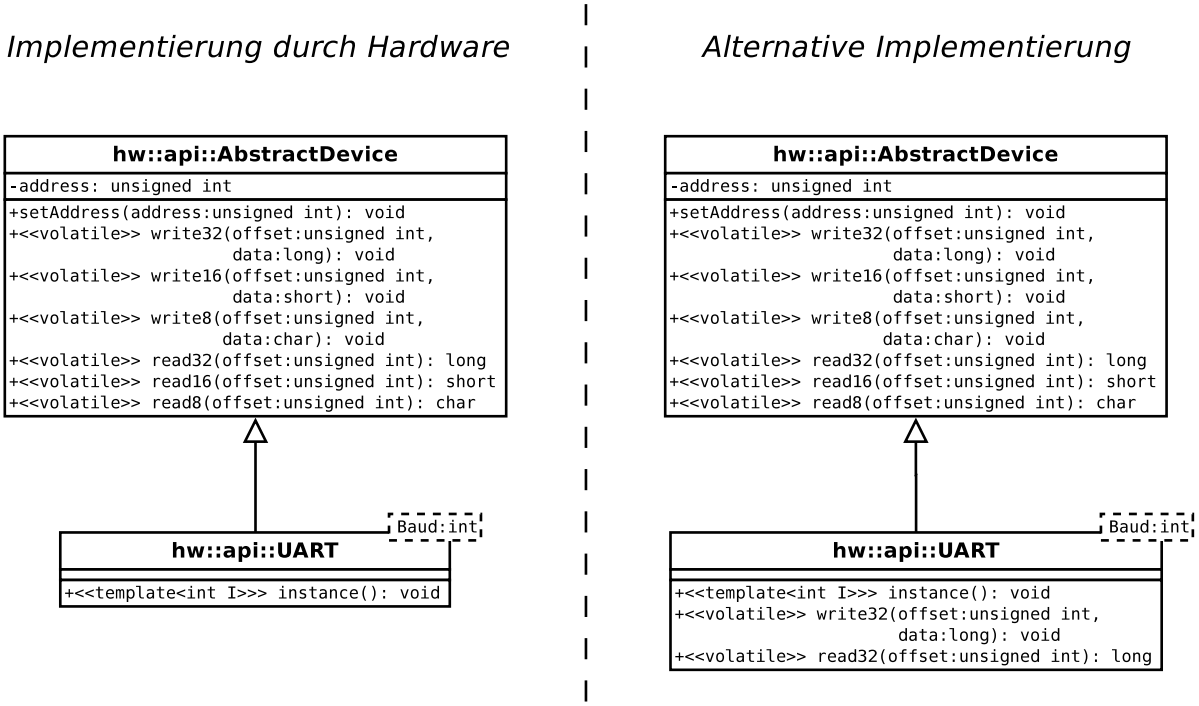

Abbildung 3.3: Unterschiedliche Implementierungen der Hardware API

Abbildung 3.3 zeigt Klassendiagramme, sowohl für die Verwendung der API auf Hardware als auch für alternative Lösungen. Unter Beibehaltung der Schnittstelle werden die Methoden der Oberklasse `AbstractDevice` durch eine direkte Implementierung in den Hardwareklassen verdeckt. Die standardmäßigen Speicherzugriffe können dann, z.B. durch Systemaufrufe von Linux, ersetzt werden.

3.3 Aufbau der Werkzeugkette

Die gesamte Werkzeugkette verfolgt das Ziel, aus einer gegebenen LavA-Anwendung ein maßgeschneidertes MPSoC abzuleiten und anschließend Hard- und Software zusammen auszuführen. In diesem Kapitel wird der interne Aufbau der Werkzeugkette näher beleuchtet. Ein erstes, grobes Schema zeigt Abbildung 3.4.

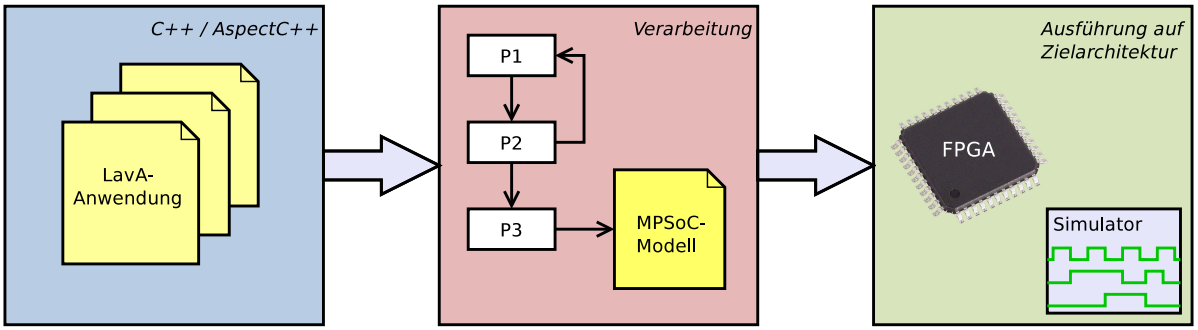

Abbildung 3.4: Skizze der Werkzeugkette

Die Arbeit gliedert sich in drei Bereiche:

1. Eingabe durch eine verteilte Anwendung
2. Verarbeitung der Anwendung zum MPSoC-Modell
3. Ausführung auf einer Zielplattform

In Kapitel 3.3.1 wird die Eingabe und ihre Vorverarbeitung erläutert. Danach folgt die Analyse und Verarbeitung in Kapitel 3.3.2. Zuletzt wird die Anwendung auf der gewünschten Zielplattform ausgeführt, was Kapitel 3.3.3 zeigt. Die gesamte Werkzeugkette ist durch ein Skript realisiert, das in Kapitel 5.5 beschrieben wird.

3.3.1 Die Eingabe der Werkzeugkette

Der gesamte Prozess benötigt als Eingabe eine Reihe von Verzeichnissen. Diese enthalten den Quelltext der verteilten LavA-Anwendung, die dann auf dem MPSoC ausgeführt wird. Dabei besteht jedes Verzeichnis aus einem eigenständigen Programm, welches seine Hardwareanforderungen durch die instanziierte Hardware API ausdrückt. Nähere Angaben zur Eingabe liefert Kapitel 4.1.1.1.

Um aspektorientierte Programmierung einzubinden, besteht der erste Arbeitsschritt aus dem Weben eventuell vorhandener Aspekte. Ein Beispiel wäre das Testen der Anwendung über textuelle Ausgaben. Um jeden Funktionsaufruf untersuchen zu können, webt man per Aspekt einen zusätzlichen UART in das System und nutzt seine Schnittstelle zur Ausgabe der benötigten Informationen. Nach dem Abschluss der Tests wird der Aspekt entfernt und passt somit die Hardwarestruktur an.

3.3.2 Analyse der Anwendung

Nachdem die verteilte Anwendung gewoben wurde, untersucht der in Kapitel 4.1 vorgestellte Parser ihren Quelltext. Dabei analysiert er die Instanziierung und Initialisierung der Hardware API. Intern wird die gewünschte Hardware lokal, d.h. nur innerhalb der SoCs, auf Konsistenz geprüft, um beispielsweise Fehler bei der Adressraumkonfigurierung auszuschließen. Aufgetretene Konflikte können teilweise durch Manipulation des Quelltextes behoben werden.
Anschließend kann die aktualisierte Anwendung innerhalb der einzelnen Verzeichnisse kompiliert werden. Die nachfolgende Zielplattform erhält als Zwischenergebnis die einzelnen Kompilate sowie das MPSoC-Modell. Abbildung 3.5 veranschaulicht den Prozess bis zu dieser Stelle.

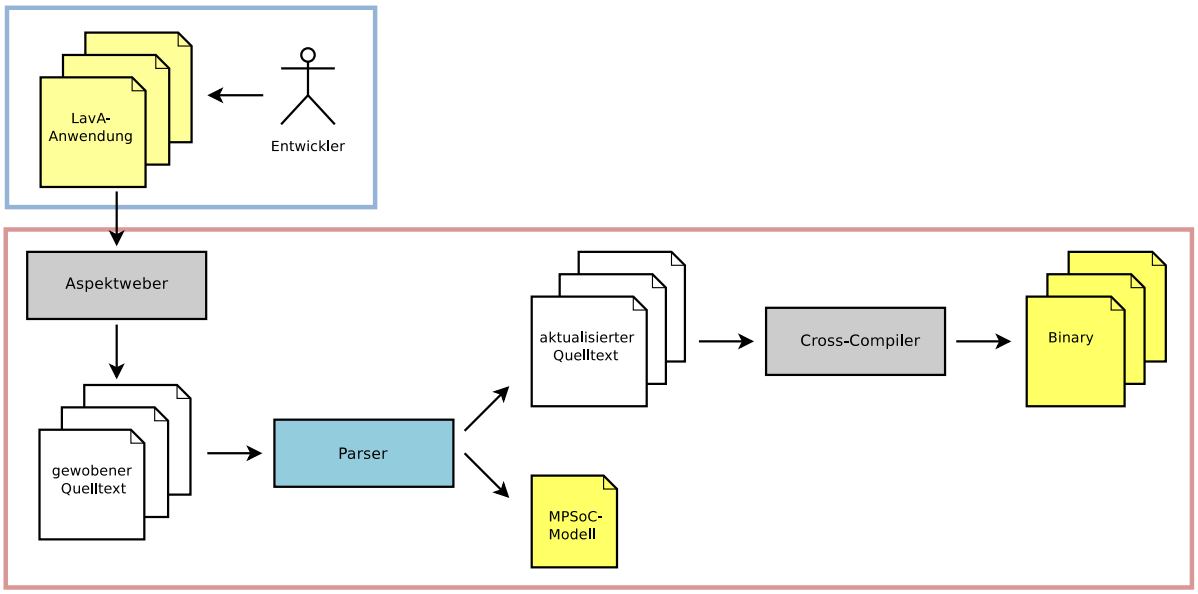

Abbildung 3.5: Von der Eingabe zum MPSoC-Modell

3.3.3 Ausführung auf einer Zielplattform

Zwei Alternativen stehen für die Auswahl der Zielplattform zur Verfügung: Die Implementierung des MPSoCs auf einem FPGA oder eine Simulation. Die notwendigen Schritte für die Hardwarelösung präsentiert Kapitel 3.3.3.1, während Kapitel 3.3.3.2 die Simulation erläutert.

3.3.3.1 Hardwaresynthese

Im Kontext dieser Arbeit besteht Hardware oder eine Hardwareplattform aus einer textuellen Hardwarebeschreibung mit entsprechenden Sprachen wie VHDL oder Verilog. Die Abbildung der Hardwarebeschreibung auf physikalische Komponenten wird Synthese genannt und erzeugt ein sog. *Bitfile*, mit dem ein FPGA geladen werden kann.

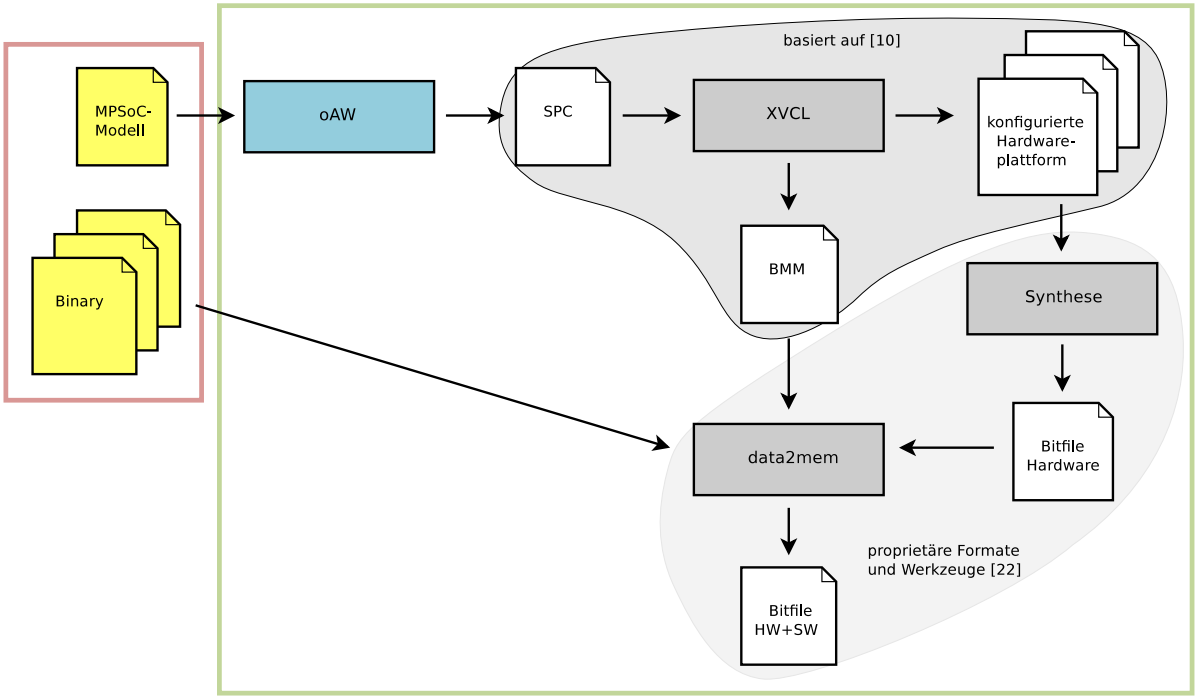

Abbildung 3.6: Werkzeugkette für die Implementierung auf Hardware

Der gesamte Ablauf wird in Abbildung 3.6 wiedergegeben. Zunächst wird mit dem MPSoC-Modell die Konfigurationsdatei der Hardwareplattform generiert. Dies geschieht mit einem oAW-Projekt, welches auf das Metamodell von Kapitel 3.2.1.1 zurückgreift. Darin integriert ist die zweite Stufe der Validierung, die das Modell SoC-übergreifend betrachtet und Fehler erkennt. Kapitel 5.4 veranschaulicht das genannte Projekt.

Danach interpretiert XVCL die Konfigurationsdatei und erzeugt die maßgeschneiderte Hardware, die durch proprietäre Programme des entsprechenden FPGA-Herstellers synthetisiert wird. Das resultierende *Bitfile* wird in einem letzten Arbeitsschritt mit den vorab erstellten Kompilaten aktualisiert. Dazu werden die Initialisierungsdaten der Speicherbausteine mit den Programmen und Daten gefüllt. Das Hard- und Softwareumfassende *Bitfile* muss letztlich auf das FPGA geladen werden.

3.3.3.2 Simulation

Der in dieser Diplomarbeit entwickelte Simulator basiert auf dem Betriebssystem Linux und führt die einzelnen SoCs als eigenständige Prozesse aus. Weiter wird Interprozesskommunikation eingesetzt, um die Peripherie der SoCs zu simulieren und ihre Vernetzung herzustellen. Dazu interpretiert der Simulator das MPSoC-Modell und erstellt die nötigen Dateien und Artefakte der Interprozesskommunikation. Die LavA-Anwendung muss für x86 übersetzt werden und enthält eine alternative Implementierung der Hardware API, die nun Systemaufrufe statt Speicherzugriffe durchführt. Weitere Informationen zum Simulator liefert Kapitel 4.2.

4 Entwurf ausgewählter Werkzeuge

Während das vorige Kapitel das Gesamtkonzept der Werkzeugkette vorstellt, soll hier der Entwurf einzelner Komponenten ausführlicher behandelt werden. Von zentraler Bedeutung ist der Parser, dessen Vorgehen in Kapitel 4.1 präsentiert wird. Anschließend stellt Kapitel 4.2 den Simulator vor, der eine Alternative zur Entwicklung mit FPGAs bieten soll. Kapitel 4.3 definiert den Begriff der Kosten für ein MPSoC und erläutert ihre Bedeutung für die Werkzeugkette.

4.1 Der Parser

Der Parser stellt das Herzstück der Werkzeugkette dar und wird in diesem Kapitel genauer betrachtet. Er verbindet die Software- und die Hardwareebene. Durch die Suche und Analyse der Hardware API wird hier ein Modell generiert, welches das anwendungsspezifische MPSoC abbildet. Somit wird die Möglichkeit geschaffen, Hardware direkt aus der Software heraus zu instanziieren. Die Konfigurierung der Anwendung, etwa über zusätzliche oder unterschiedliche Gerätetreiber, beeinflusst gleichzeitig die benötigte Hardware und erlaubt so die durchgängige Maßschneiderung von Hard- und Software.

Die wesentliche Arbeit besteht in der Suche nach der Hardware API. Hierzu wurde auf der Parser-Bibliothek Puma (siehe Kapitel 2.2) aufgebaut, welche die Programmiersprachen C, C++ sowie AspectC++ verarbeiten kann. Dies ist notwendig, denn, wie Kapitel 3.2 zeigt, beruht die entwickelte Hardware API auf C++ *Templates*. Demnach müssen alle Anwendungen ebenfalls in C++ geschrieben werden, wie es bei eingebetteten Systemen häufig der Fall ist.
Neben der Generierung des MPSoC-Modells, muss der Parser in der Lage sein, die vorgefundene Anwendung derart zu manipulieren, dass die Interaktion mit der später fertiggestellten Hardware fehlerfrei funktioniert. Dies betrifft die Schnittstellen zwischen Hard- und Software, die entsprechend mit gleichen Unterbrechungsnummern oder Speicheradressen versorgt werden müssen.
Um die Funktionalität zu veranschaulichen, geben die folgenden Kapitel einen Überblick über die zentralen Aspekte des Parsers. Zunächst stellt Kapitel 4.1.1 die Suche nach der Hardware API vor. Anschließend wird in Kapitel 4.1.2 die Datenhaltung der gefundenen Instanzen gezeigt. Kapitel 4.1.3 präsentiert die notwendige Manipulation der Anwendung, um o.g. Konflikte beheben zu können. Schließlich erläutert Kapitel 4.1.4 die Generierung des MPSoC-Modells sowie die finale Überprüfung nach Fehlern, um dessen syntaktische Korrektheit sicherzustellen.

4.1.1 Suche und Analyse der Hardware API

Bevor die Instanzen der Hardware API gesucht werden können, werden Anforderungen an die Organisation und Aufteilung der Anwendung formuliert. Weiter muss sichergestellt werden, dass PUMA, abhängig vom Prozessor, passend eingestellt wird. Sind diese Punkte erfüllt, kann eine fehlerfreie Suche nach den einzelnen Bestandteilen durchgeführt werden.

4.1.1.1 Anforderungen an die LavA-Anwendung

Neben der Nutzung der Hardware API, werden hier weitere Anforderungen an die LavA-Anwendung gestellt. Der Parser startet in einem Quellverzeichnis, welches durch einen Parameter angegeben wird. Innerhalb dessen wird jedes Unterverzeichnis als Programm für ein entsprechendes SoC interpretiert. Daher muss die verteilte Anwendung sich über mehrere Verzeichnisse erstrecken. Quelltext 4.1 zeigt die geforderte Ordnerstruktur an. Dieser Ansatz stellt eine flexible Schnittstelle für zukünftige Erweiterungen bereit. Ähnlich zu der Arbeit, welche in Kapitel 2.4.1 skizziert wurde, wäre ein Verfahren denkbar, das eine Anwendung erst parallelisiert und anschließend in einzelne Verzeichnisse unterteilt.
Die nächste Anforderung bezieht sich auf die prozessorabhängige Konfiguration von PUMA. Um heterogene Multiprozessorsysteme zu unterstützen, muss PUMA SoC-spezifisch konfiguriert werden können. Angaben zum Prozessor, etwa die Bitbreite einzelner Datentypen, sind ebenso notwendig, wie *Include*-Pfade. Letztere ermöglichen erst eine fehlerfreie Analyse des vorliegenden Quelltextes. PUMA kann dazu Konfigurierungsdateien einlesen, die die genannten Informationen zusammenfassen.

```
...
-I "/absolute/path/to/libraries/hwapi"
-p "/absolute/path/to/project/mpsoc/soc1/source"
-d "/absolute/path/to/project/mpsoc/soc1/source"
```

Quelltext 4.1: SoC-spezifische Verzeichnisse in PUMAs Konfigurationsdatei

Pro SoC muss eine solche Datei, benannt „parser.config“, im jeweiligen Unterverzeichnis des SoCs enthalten sein. Die nötigen Daten der Prozessoren kann PUMA automatisch aus dem passenden Übersetzer laden und direkt in einer Konfigurationsdatei ablegen. Dies geschieht mit dem Aufruf aus Quelltext 4.2.

```
$ ag++ --c_compiler <target-compiler> --gen_config -o parser.config
```

Quelltext 4.2: Generierung einer initialen „parser.config“

Anschließend müssen noch *Include*-Pfade und PUMAs Projektpfade hinzugefügt werden. Erstere werden durch den Parameter `-I` eingeleitet. Die Projektpfade werden mit `-p` und `-d` spezifiziert und geben die Verzeichnisse an, in dem PUMA nach Quelltext suchen sowie manipulierten Quelltext ablegen soll. Beide Argumente zielen auf das oberste Programmverzeichnis des jeweiligen SoCs. Quelltext 4.1 enthält einen Auszug der Datei „parser.config“ für das SoC1, wobei der Quelltext im letzten Ordner anzufinden wäre.

Der Parser startet in diesem Beispiel in dem Verzeichnis „mpsoc“. Zusätzlich zu dem ersten Unterverzeichnis „soc1“, gibt es mit „source“ noch ein zweites Unterverzeichnis, in dem sich letztlich der Quelltext befindet. Die in Kapitel 4.1.3 vorgestellten Techniken zur Manipulation der Anwendung benötigen diese zweite Unterebene, falls das Programm Unterbrechungen nutzt.

4.1.1.2 Suche der Hardware API Instanzen

Wie im vorigen Kapitel erwähnt, bezieht sich die Suche auf gewobene Quelltextdateien. Weiterhin werden die möglichen Instanziierungen der Hardware API in Kapitel 3.1.4 durch globale Objekte definiert und in Kapitel 3.2.2 durch Beispiele vorgestellt.
Der hier beschriebene Algorithmus wird im Parser durch die Klasse `ApiParser` implementiert. Zunächst wird über alle gewobenen Übersetzungseinheiten iteriert, die PUMA erst syntaktisch, dann semantisch analysiert. Die semantischen Informationen werden in der sog. semantischen Datenbank `CSemDatabase` zusammengetragen. Die einfachste Klassifizierung dieser Informationen stellt die Klasse `CObjectInfo` bereit. Spezialisierungen enthalten dann zusätzliche Angaben, u.a. besitzt `CClassInfo` Daten zu Klassendefinitionen und damit auch Verweise auf mögliche Oberklassen.
Nachdem die Übersetzungseinheit analysiert wurde, werden alle Attribute und Funktionen begutachtet, ob sie selbst Instanzen der Hardware API sind oder jene enthalten. Attribute umfassen jegliche Variablendeklarationen und Klassenattribute. PUMA ordnet diesen jeweils ein Objekt der Klasse `CAttributeInfo` zu. Funktionen werden durch die Klasse `CFunctionInfo` repräsentiert. Der Pseudocode in Quelltext 4.3 veranschaulicht den Suchalgorithmus.
Die Suche der Hardware API Instanzen erstreckt sich nicht nur auf direkte Objekte, sondern analysiert deren gesamte Typhierarchie und Attribute. In den Zeilen 22–25 werden die Oberklassen untersucht, von denen das globale Objekt, bzw. dessen zugehörige Klasse, erbt. Die Zeilen 26–32 verarbeiten mögliche Klassenattribute, denn diese können ebenfalls eine Instanz enthalten. Durch die gezeigte Struktur sind auch Kombinationen von Vererbung und Komposition in beliebiger Reihenfolge und Anzahl möglich.
Hauptbestandteil der Suche sind Listen. Diese enthalten als letztes Element stets Informationen über die aktuell zu untersuchende Klasse. Für die Bearbeitung von Oberklassen wird dieser Eintrag einfach ausgetauscht und die Liste erneut analysiert. Bei vorgefundenen Klassenattributen werden die Informationen zu dem Attribut sowie dessen Klasse gespeichert. Dadurch sind alle Daten der Hardware API Instanz verfügbar und können zusammen gesichert werden. In Kapitel 4.1.2 wird diese Datenstruktur detaillierter behandelt.
Wird während der Ausführung des Algorithmus' zunächst ein Objekt der Klasse `hw::irq::IRQ` gefunden, wird die anschließend gefundene Hardware API Instanz mit geforderter Unterbrechung abgespeichert. Die Klasse `hw::irq::IRQ` ist selbst ein *Template* und enthält einzig die spätere Unterbrechungsnummer. Vorab ist diese mit -1 vorinitialisiert. Kapitel 4.1.3.2 schildert, wie sie neu berechnet und in der Anwendung verankert wird. Quelltext 4.4 zeigt eine Typhierarchie mit Unterbrechung.

```
Suche:
  Iteriere über Dateien mit Endung .acc
    Analysiere diese Datei syntaktisch und semantisch
    Iteriere über alle Attribute
      Falls Attribut A ein globales Objekt
            oder statisches Klassenattribut ist
        Untersuche Objekt A
    Iteriere über alle Funktionen
      Falls Funktion F eine Singleton-Methode ist
        Untersuche Objekt F

Untersuche Objekt (Objekt O):
  Ermittle Klasse K von O
  Erstelle Liste L = (O, K)
  Untersuche Liste L

Untersuche Liste (Liste L):
  Hole letztes Element von L als Klasse K
  Falls K eine Klasse der Hardware API ist
    speichere Liste L als Instanz ab
  sonst
    Iteriere über alle Oberklassen von K
      Erstelle Kopie L' der Liste L
      Tausche in L' Klasse K mit Oberklasse K'     // L' = (O, K')
      Untersuche Liste L'
    Iteriere über alle Klassenattribute von K
      Erstelle Kopie L' der Liste L
      Entferne Klasse K aus L'                     // L' = (O)
      Füge Klassenattribut A der Liste L' hinzu    // L' = (O, A)
      Ermittle Klasse K' von A
      Füge Klasse K' Liste L' hinzu                // L' = (O, A, K')
      Untersuche Liste L'
```

Quelltext 4.3: Algorithmus für die Suche der Hardware API Instanzen

```
class DebugPort {
  // Zeigt geforderte Unterbrechung des UARTs an
  hw::irq::IRQ<DebugPort> irq;
  hw::api::UART<57600> uart;
};
```

Quelltext 4.4: Hardware API Instanz mit Unterbrechung

4.1.1.3 Suche der Initialisierungen

Initialisierungen sind Aufrufe der *Template*-Funktion `instance`, die jede Peripherie der Hardware API zur Verfügung stellt. Zum einen berechnet die Methode die Speicheradresse, an der das Gerät im Adressraum des Prozessors zu finden ist. Zum anderen wird der übergebene *Template*-Parameter, eine Ganzzahl, vom Parser an späterer Stelle analysiert.

```
// Instanziierter CAN Controller
hw::api::CAN<> can;

void init() {
  // Initialisierung des CAN Controllers
  can.instance<0>();
}
```

Quelltext 4.5: Instanziierung und Initialisierung eines CAN Controllers

Die Zuordnung der Initialisierungen zu den passenden Hardware API Instanzen wird in Kapitel 4.1.2 erläutert. Kapitel 4.1.3 enthält u.a. die Korrektur der hier vorgefundenen *Template*-Parameter, um eventuelle Konflikte zwischen unterschiedlichen Instanzen zu beheben. Daher müssen die Syntaxteilbäume zu den *Template*-Parametern gefunden und gespeichert werden.
Im Laufe der Entwicklung wurden zwei Ansätze verfolgt, die Initialisierungen zu finden. Zum einen die semantische Analyse aller *Template*-Funktionen einer Übersetzungseinheit. Zum anderen eine Traversierung des Syntaxbaums. Beide Strategien werden in eigenen Kapiteln vorgestellt und abschließend bewertet.

Semantische Analyse von Template-Funktionen

Die Hardware API Instanzen werden durch semantische Informationen der Übersetzungseinheiten gefunden. Dieser Ansatz ist sehr komfortabel, denn PUMA organisiert diese Daten innerhalb der Klasse `CSemDatabase`, wodurch sie leicht zugänglich sind. Ebenso möglich ist eine Iteration über alle *Template*-Funktionen, welche in PUMA durch die Klasse `CFctInstance` modelliert werden.
Daher lässt sich ein Algorithmus, ähnlich zu Quelltext 4.3, formulieren, der die Initialisierungen herausfiltert. Dazu werden Name und Signatur der Methode überprüft. Letztere muss genau einen *Template*- und keinen Funktionsparameter aufweisen. Der *Template*-Parameter muss zusätzlich vom Typ `int` sein. Danach wird das Objekt, auf dem die Methode aufgerufen wurde, daraufhin untersucht, ob dessen Klasse aus der Hardware API stammt.
Sind alle Voraussetzungen erfüllt, wird der Aufruf, als gefundene Initialisierung, gespeichert. Der Syntaxbaum des *Template*-Parameters ist über den Verweis `Tree` erreichbar, der in jedem semantischen Objekt, also auch in `CFctInstance` enthalten ist.

Traversierung des Syntaxbaums

Neben der Herangehensweise über die semantische Ebene, kann PUMA einen Quelltext auf syntaktischer Ebene bearbeiten. Der Syntaxbaum besteht aus Knoten der Klasse `CTree`, welche sowohl Verweise auf `Token` als auch `CSemObject` besitzen und damit die Verbindung zwischen Zeichen des Quelltextes und ihren semantischen Objekten herstellen.

In Baumstrukturen wird über Elter-, Kind- und Nachbarknoten iteriert, was eine umständliche Schnittstelle darstellt. PUMA schafft hier jedoch Abhilfe über die Klasse `CVisitor`, welche bereits eine Tiefensuche implementiert. Eigene Funktionalität wird über eine Spezialisierung dieser Klasse erreicht, welche die Methoden `pre_visit` und `post_visit` konkretisieren.

Die Klasse `InitVisitor` implementiert die Suche der Initialisierungen über den Syntaxbaum. Wie beschrieben, traversiert die Oberklasse `CVisitor` durch den Baum und ruft für jeden Knoten `pre_visit` auf. Zunächst werden alle Knoten außer `CT_CallExpr`-Knoten ignoriert. Diese stehen für Anweisungen, die einen Aufruf enthalten, etwa den `instance`-Aufruf. Folgender Pseudocode skizziert die Funktionalität von `pre_visit`.

```
pre_visit(CTree T):
  Falls T vom Typ CT_CallExpr ist
    Falls aufgerufene Funktion eine Template-Funktion ist
      Falls Aufruf Template-Funktion instance ist
        Merke semantische Information F zum Aufruf
  Falls F gesetzt ist
    Falls T vom Typ CT_TemplateArgList ist
      Speichere Aufruf F mit Template-Parameter T
      Lösche F
```

Quelltext 4.6: Funktionalität von `pre_visit`

Nachdem ein passender Knoten gefunden wurde, wird dessen semantische Information zum Methodenaufruf gespeichert. Diese Variable wird innerhalb von `InitVisitor` gesichert und steht damit bei jeder Ausführung von `pre_visit` übergreifend zur Verfügung. Wird daraufhin ein Syntaxknoten gefunden, der auf den *Template*-Parameter schließen lässt, kann die Verknüpfung von Aufruf und *Template*-Parameter gespeichert werden. Dank der verwendeten Tiefensuche gehört der gefundene *Template*-Parameter stets zum aktuell gespeicherten Methodenaufruf.

Bewertung der Alternativen

Die Semantische Analyse von *Template*-Funktionen bildet eine komfortable Lösung, die der Suche nach den Hardware API Instanzen gleicht. Daher wäre eine Kapselung beider Verfahren innerhalb der Klasse `ApiParser` möglich. Der nächste Vorteil wäre die gemeinsame Iteration über semantische Objekte, die den Startpunkt für beide Algorithmen bildet.

Diese Strategie liefert jedoch nur eine Teilmenge der nötigen Informationen: Würde der CAN Controller in Beispiel 4.5 ein zweites Mal, mit identischem Parameter initialisiert werden, dann wäre der zweite Aufruf nicht in der Lösungsmenge enthalten. PUMA erzeugt lediglich für die jeweils ersten Vorkommen semantische Objekte. Diese Objekte besitzen zudem nur den Verweis auf ihr erstes Auftreten im Syntaxbaum.

Aus diesem Grund wurde der zweite Ansatz im Parser implementiert. Das Verfahren über die syntaktische Ebene findet jede Initialisierung, auch wenn mehrere von ihnen auf das selbe semantische Objekt verweisen. So wird zu einem Methodenaufruf eine Liste von *Template*-Parametern in Form von Syntaxbäumen gespeichert, um alle notwendigen Informationen zusammenzutragen.

4.1.2 Datenhaltung der Instanzen

Die Aufgabe der Datenhaltung ist die strukturierte Speicherung der Hardware API Instanzen und ihrer Initialisierungen. Da beide Teilinformationen zu unterschiedlichen Zeiten gesichert werden, müssen sie nachträglich miteinander verknüpft werden können. Weiterhin sollten die Daten effizient zugreifbar sein, da jede weitere Arbeit des Parsers auf diesen Daten beruht.

Die Kapselung der Daten wurde in zwei Klassen aufgeteilt. Die Klasse `HwApiInstance` bündelt – wie der Name besagt – alle Informationen zur Hardware API Instanz. Die Objekte dieser Klasse werden in dem `DataContainer` hinterlegt, welcher einmal pro SoC existiert. Abbildung 4.1 veranschaulicht den hier relevanten Teil der beiden Klassen.

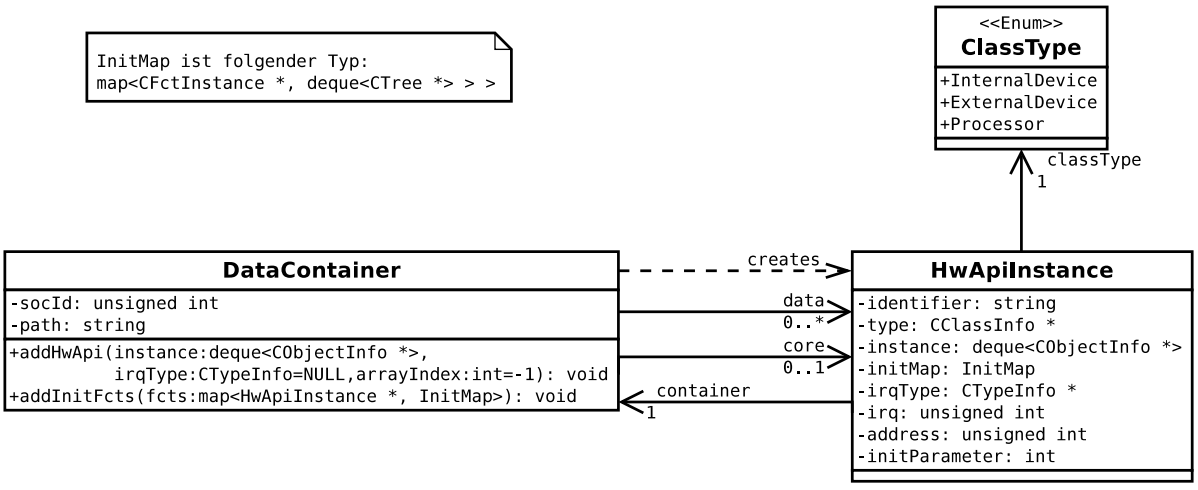

Abbildung 4.1: Modellierung der Datenhaltung im Parser

Bevor die Klassen einzeln vorgestellt werden, soll zunächst der textuelle *Identifier* beschrieben werden, der die Zuordnung zwischen Instanz und Initialisierung herstellt.

4.1.2.1 Der Schlüssel einer Instanz

Als Schlüssel der `HwApiInstance`-Klasse wurde ein textueller *Identifier* eingesetzt. Die in Kapitel 4.1.1.2 vorgestellte Liste dient als Eingabe für neue Instanzen im Container. Sie beinhaltet alle vorgefundenen Attribute, bis hin zum eigentlichen Objekt, dessen Klasse aus der Hardware API stammt. Die Namen der Attribute werden durch Punkte verbunden und ergeben den Schlüssel. Gleichzeitig stellt der Schlüssel eine gültige C++ Referenzierung des letzten Objekts dar, falls jedes Attribut öffentlich deklariert wurde. Für das erste Attribut muss der Name inklusive der zugehörigen Namensräume aufgeführt werden. Es folgen zwei Beispiele.

```
namespace hw {
  namespace dev {
    class Test1 : public hw::api::UART<> {
    } test1;              // Schlüssel: "hw::dev::test1"
  }

  class Test2 {
  public:
    hw::api::CAN<> can;   // Schlüssel: "hw::test2.can"
  } test2;
}

void init() {
  // Initialisierung nutzt die Schlüssel
  hw::dev::test1.instance<0>();
  hw::test2.can.instance<0>();
}
```

Quelltext 4.7: Beispiele für den Schlüssel einer Hardware API Instanz

Die Initialisierungen werden dem Container später hinzugefügt. Der Schlüssel muss direkt im Aufruf der Methode `instance` enthalten sein, sodass der `InitVisitor` ihn aus dem Syntaxbaum heraus zusammenfügen kann.
Somit wird die Verknüpfung zwischen Instanz und Initialisierung über den Textvergleich beider Schlüssel hergestellt. Problematisch ist die Mehrfachvererbung von mehreren Hardwareklassen. Hierbei gilt der gleiche Schlüssel für alle Instanzen. Daher wurde ein Vergleich der Hardwareklassen hinzugefügt.

4.1.2.2 Die Klasse DataContainer

Für die Verwaltung der Instanzen wird der *STL*-Container `set` eingesetzt. Damit werden die Hardware API Instanzen einmalig und eindeutig gesichert. Ein `set` wird intern durch eine Baumstruktur abgebildet. Neue Elemente werden anhand ihres Schlüssels passend im Baum eingefügt. Die Baumstruktur ermöglicht dann einen effizienten Zugriff auf ihre Elemente.

Der `DataContainer` ist die primäre Schnittstelle für die Datenhaltung. Die bereits beschriebenen Suchalgorithmen nutzen die Methoden `addHwApi` sowie `addInitFcts`, um die Instanzen bzw. Initialisierungen der Hardware API abzulegen. Der `DataContainer` erstellt dann neue Objekte der Klasse `HwApiInstance` und legt die gefundenen Informationen darin ab.
Als weitere Daten sichert der Container die Nummer des SoCs, für welches er die Instanzen sammelt, sowie den Dateipfad, der zu dem Unterverzeichnis des SoCs führt. Zusätzlich wird ein direkter Verweis auf die Prozessorinstanz gespeichert. Dieser wird für die Organisation des Adressraums benötigt.

4.1.2.3 Die Klasse HwApiInstance

Diese Klasse kapselt die eigentlichen Informationen einer Instanz. Sie enthält zuerst die Instanziierung und die Initialisierung(en). Später werden die Speicheradresse und Unterbrechungsnummer hinzugefügt. Weiter wird direkt eine Klassifizierung der Instanz bestimmt. Diese ist in Kapitel 3.2.1.3 unter dem Attribut `_ClassType` erklärt.
Eine Anwendung kann signalisieren, dass ein Gerät mit Unterbrechungen betrieben werden soll. Dazu ist eine Instanziierung des *Templates* `IRQ`, wie in Beispiel 4.4, nötig. Diese Instanz wird in der `HwApiInstance` als Attribut `irqType` gespeichert. Nachdem alle Hardware API Instanzen gefunden wurden, vergibt der Parser die wahre Unterbrechungsnummer, falls das Attribut ungleich `NULL` ist.

4.1.3 Manipulation der Anwendung

Hardware und Treiber müssen zueinander passen. Dies betrifft die verwendeten Speicheradressen und Unterbrechungsnummern. Beide Angaben hängen von der Struktur des MPSoCs ab, welche erst der Parser vollständig erfassen kann. Daher müssen Maßnahmen getroffen werden, die diese Daten nachträglich in der Anwendung einfügen. Hierzu beschreibt Kapitel 4.1.3.1, wie fehlerhafte Initialisierungen nachträglich korrigiert werden können. Anschließend wird in Kapitel 4.1.3.2 gezeigt, wie die Unterbrechungsnummern in der Anwendung platziert werden. Die hier vorgestellten Inhalte wurden in der Klasse `HwConf` verwirklicht.

Eine nachträgliche Anpassung und Ergänzung des Quelltextes erfordert die erneute Übersetzung des Quelltextes. Da die Änderungen im gewobenen Programmcode stattfinden, müssen nur diese Dateien kompiliert und zu einem Kompilat zusammengebunden werden. Ein erneutes Weben ist dagegen nicht erforderlich.

4.1.3.1 Korrektur der Initialisierungen

Die LavA-Anwendung enthält stets Interaktion mit der Hardware. Dies wird über Speicherzugriffe realisiert. Damit die Anwendung bzw. ein Treiber mit den passenden Speicheradressen kommunizieren kann, wird vorab die Initialisierung der Hardware API Instanz ausgeführt. Hierbei können Fehler auftreten, die der Parser weitestgehend korrigieren kann.

Die Anfangsadresse eines Gerätes wird über ihre Instanznummer berechnet, die bei der Initialisierung mit Hilfe der `instance`-Methode berechnet wird. Dabei ist sie abhängig von dem dort spezifizierten *Template*-Parameter. Mögliche Fehler umfassen eine fehlende Initialisierung sowie eine mehrfache und unterschiedliche Initialisierung.
Die fehlende Initialisierung kann nicht nachträglich eingefügt werden. Der Entwickler wird daher aufgefordert dies manuell zu erledigen und danach die Werkzeugkette neu zu starten.
Eine unterschiedliche Initialisierung wird jedoch vom Parser behandelt. Der Mehraufwand bei der Suche aller Initialisierungen im Quelltext (siehe Kapitel 4.1.1.3), kommt hier zum Tragen. Zunächst werden alle gefundenen Parameter eingelesen und als Hilfestellung auf der Konsole ausgegeben. Der Entwickler wird dann aufgefordert eine neue Belegung einzugeben. Letztlich wird der Quelltext manipuliert und enthält ausschließlich den neuen Parameter für alle Initialisierungen der entsprechenden Hardware API Instanz. Parallel dazu wird die endgültige Adresse im MPSoC-Modell gespeichert.
Manipulationen sind ein Kerngebiet von Puma, welches sowohl Änderungen an der Symbolkette, als auch auf Syntaxebene erlaubt. In diesem Fall wird der neu eingegebene Parameter in einen Syntaxbaum überführt. Änderungen nimmt der `ManipCommander` von Puma vor, der hier die alten Syntaxbäume durch den neuen austauscht.

4.1.3.2 Unterbrechungsnummern

Die Geräte der Hardware API werden standardmäßig ohne Unterbrechungen konfiguriert. Wird während der Suche eine Instanziierung der *Template*-Klasse `hw::irq::IRQ` gefunden, zeigt dies eine gewünschte Unterbrechungsnummer an. Diese Klasse kapselt die eigentliche Unterbrechungsnummer. Dabei wird sie jeweils spezialisiert, um unterschiedliche Nummern zu repräsentieren. Siehe dazu auch die Kapitel 5.4.4 und 6.6.2 in [26]. Der folgende Quelltext 4.8 zeigt die eigentliche Klassenimplementierung sowie eine mögliche Spezialisierung.

```
namespace hw {
namespace irq {
  template<typename T>
    struct IRQ {
      enum { NUM = -1 };
      int num() { return NUM; }
    };

  // ab hier automatisch generiert
  template<>
  struct IRQ<DebugPort> {
    enum { NUM = 0 };
    int num() { return NUM; }
  };
}}
```

Quelltext 4.8: Implementierung der Klasse `hw::irq::IRQ`

Das Beispiel ergänzt Quelltext 4.4. Dort wurde ein Objekt `irq` der Klasse `hw::irq::IRQ<DebugPort>` instanziiert, welches nun die Unterbrechungsnummer 0 enthält. Technisch gesehen werden die beiden Klassen getrennt voneinander gespeichert. Während Erstere in der Anwendung definiert wird, erzeugt erst der Parser die Spezialisierung. Diese wird in der Headerdatei „IRQ_generated.h“ im Unterverzeichnis des aktuellen SoCs abgelegt. Daher muss in der Anwendung eine `#include`-Anweisung diese Datei einbinden. Der Parser überschreibt diese und bindet so die Unterbrechungsnummern, passend zum generierten MPSoC-Modell, in der Anwendung ein.

4.1.4 Modellgenerierung

Der letzte Arbeitsschritt ist die Abbildung der MPSoC-Struktur durch ein Modell. Das Modell wird mit *libxml2* [27] in der Klasse `ModelGenerator` des Parsers erzeugt. Dabei wird das spezielle Format *XML Metadata Interchange* [28] eingehalten. Der Parser iteriert an dieser Stelle über alle vorhandenen `DataContainer`. Anschließend werden die jeweils enthaltenen Hardware API Instanzen durch XML-Knoten mit entsprechenden Attributen und Referenzen gespeichert. Quelltext 4.9 zeigt ein generiertes Modell.

```
<?xml version="1.0"?>
<MPSoC xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:xmi="http://www.omg.org/XMI"
       xmlns="hw::api"
       xmi:version="2.0"
       xsi:schemaLocation="hw::api metamodel.ecore">
  <ConnList SoCList="//@SoCList[Id='1'] //@SoCList[Id='2']" Id="1"
            Type="Bus" Width="32" MessageQueue="5" OutBuffer="5"/>
  <SoCList Id="1">
    <_CPU xsi:type="MIPS" RAM="16"/>
    <_Devices xsi:type="IPC" _Base="2281703424"
                             Bus="//@ConnList[Id='1']"/>
    <_Devices xsi:type="IRQController" _Base="2147483648"/>
  </SoCList>
  <SoCList Id="2">
    <_CPU xsi:type="ZPU" RAM="8"/>
    <_Devices xsi:type="IPC" _Base="2281703424"
                             Bus="//@ConnList[Id='1']"/>
    <_Devices xsi:type="UART" _Base="2148536320" _UCF="0"
                              Baud="57600"/>
    <_Devices xsi:type="IRQController" _Base="2147483648"/>
  </SoCList>
</MPSoC>
```

Quelltext 4.9: Beispiel eines MPSoC-Modells

Die hier dargestellte Struktur entspricht dem Metamodell aus Kapitel 3.2.1.1. Das MPSoC besitzt zwei SoCs, die über einen Bus verbunden sind. Weiter sind Interrupt Controller und ein UART enthalten. Die Prozessoren bestehen aus einem MIPS mit 16KB RAM und einer ZPU mit 8KB RAM.

Besonderer Wert muss bei der Generierung auf die verschiedenen Identifikationsnummern gelegt werden. Mit ihnen referenzieren IPC Controller das jeweils angeschlossene Netzwerk. Ebenso verweist eine Kommunikationsstruktur auf die SoCs, die verbunden werden sollen. Stimmen diese Nummern nicht überein, kann oAW das Modell nicht einlesen. Der Parser überwacht daher die gefundenen Identifikationsnummern und zeigt fehlende dem Entwickler an.

4.2 Der Simulator

Der für LavA geplante Simulator soll eine Anwendungsentwicklung ohne benötigte Hardware ermöglichen. Die Hardware und ihre Ansteuerung muss dazu simuliert werden, sodass die eigentliche Anwendung ohne Änderungen übernommen werden kann. Das Ziel der Simulation ist es, eine zeiteffiziente Alternative zu bieten, die gleichzeitig weniger Fehlerquellen enthält.

Kapitel 4.2.1 erläutert, wie die einzelnen Komponenten des MPSoCs durch eine Simulation mit Linux realisiert werden können. Ihre Umsetzung erfolgt in einer zweiten Version der Hardware API, wie Kapitel 3.2.3 erklärt. Die Koordination der einzelnen Komponenten übernimmt der eigentliche Simulator, welcher in Kapitel 4.2.2 vorgestellt wird.

4.2.1 Abbildung des MPSoCs auf Linux

Hardwarebeschreibungen für FPGAs werden normalerweise mit entsprechenden Simulatoren wie Xilinx' *ISim* [25] oder einer *Open Source* Alternative, bestehend aus *GHDL* [29] und *GTKWave* [30], getestet. Das System wird über eine gewisse Zeitspanne simuliert. Die gewonnenen Ergebnisse werden durch Signalverläufe und Registerbelegungen visualisiert. Die resultierende Abstraktionsebene ist eine andere, als wenn ein Programm mit `printf`-Ausgaben analysiert wird. Letzteres reicht meist jedoch aus und führt schneller zum Ziel.

Die Abstraktionsebene sollte also möglichst hoch sein, damit der direkte Bezug zwischen Anwendung und Simulation gegeben ist. Daher bietet sich eine Implementierung durch Linux-Prozesse an. Ein SoC wird durch einen eigenständigen Prozess abgebildet, der dann die ihm zugeteilte Anwendung ausführt. Demnach erhält man aus einem MPSoC eine Menge von Prozessen. Der Nachteil ist das nicht vorhersagbare Zeitverhalten einer Simulation. Dies ist abhängig vom *Scheduler*, von der Anzahl der verfügbaren Prozessorkerne und von der jeweiligen Auslastung des Systems.

In den folgenden Kapiteln wird die Abbildung der MPSoC-Komponenten durch Linux-Betriebssystemmittel einzeln geschildert. Von besonderer Bedeutung ist hier die Interprozesskommunikation, die Linux wie folgt bereitstellt:

- Dateiein-/ausgaben
- *Pipes* bzw. *FIFOs*
- *Message Queues*
- Gemeinsamer Speicher
- *Sockets*
- Signale

4.2.1.1 IPC Controller

Die Vernetzung der SoCs wird mit dem *IPC Controller* hergestellt. Für die Kommunikation sind die Identifikationsnummer des Ziels und die Daten anzugeben. Adressiert werden können lediglich SoCs. Alle an dem Bus angeschlossenen SoCs erkennen, dass eine Nachricht vorliegt, ignorieren diese jedoch, falls ihre Identifikationsnummer nicht mit der Zieladresse übereinstimmen.
Message Queues ähneln diesem Prinzip, denn mehrere Prozesse können sich mit ihnen verbinden und unaufgefordert Nachrichten abschicken. Die Nachricht erhält neben dem eigentlichen Inhalt eine Priorität oder Klassifizierung durch eine Nummer. Analog kann der Empfang auf Nachrichten einer bestimmten Nummer begrenzt werden. Somit bilden *Message Queues* eine adäquate Möglichkeit, das MPSoC aufzubauen.

4.2.1.2 Unterbrechungen

Unterbrechungen werden durch die Geräte innerhalb eines SoCs erzeugt. Der Zeitpunkt hängt von ihrer jeweiligen Aktivität ab und ist für den Prozessor nicht vorhersehbar. Anschließend kann dieser die Unterbrechungen abarbeiten, indem der *Interrupt Controller* die Geräte bekanntgibt, welche eine Unterbrechung fordern.
Solch asynchrone Unterbrechungen sind für Linux-Prozesse durch Signale implementiert Der Prozess muss zuvor bekanntgeben, ob und wenn ja, welche Signale er erhalten möchte. Danach kann jeder beliebige Prozess ein Signal erzeugen und damit eine Unterbrechung des normalen Programmflusses auslösen, ähnlich zur richtigen Hardware.

4.2.1.3 UART und CAN

Die Geräte UART und CAN sind serielle Schnittstellen, die in LavA lediglich mit Komponenten jenseits des FPGAs kommunizieren können. Eine interne Verschaltung, wie mit dem *IPC Controller*, ist nicht möglich.
Verzichtet man auf die serielle Verarbeitung und gibt direkt die gesamten Daten aus, so lassen sich die beiden Geräte durch *FIFOs* realisieren. *FIFOs* sind benannte *Pipes*, die im Linux-Dateisystem verankert werden. Das simulierte Gerät schreibt Daten in die *FIFO*, die dann sogar an echter Hardware angeschlossen werden kann.

4.2.1.4 Timer

Der Timer bietet dem MPSoC ein prozessorunabhängiges Zählerregister und kann damit als externer Taktgeber fungieren. Regelmäßige Unterbrechungen können ebenfalls generiert werden. Dazu wird ein Vergleichswert definiert. Hat das Zählerregister diesen erreicht, meldet der Timer dies dem *Interrupt Controller*. Die in Linux weitgehend implementierte *POSIX*-Schnittstelle bietet hier einen Timer an, der genau den Anforderungen entspricht.

4.2.1.5 IO-Port

Die Ein- und Ausgabe-Ports bieten eine simple Schnittstelle, um LEDs oder Schalter anzusteuern. Implementiert sind sie durch ein Register, deren Bits eine direkte Verbindung mit Pins des FPGAs besitzen. Simuliert werden können diese Ports durch gemeinsamen Speicher. Die Änderungen darin werden direkt sichtbar, sowohl für das SoC als auch für externe Programme. Letztere könnten beispielsweise eine grafische Oberfläche enthalten, um den Zustand zu visualisieren.

4.2.2 Vernetzung durch den Simulator

Die hier angestrebte Lösung benötigt einen Hauptprozess, der die Betriebsmittel bereitstellt, bevor die einzelnen Geräte diese anfordern. Anschließend muss er für die notwendige Koordination der einzelnen SoCs sorgen.

Der erste Arbeitsschritt ist das Einlesen des MPSoC-Modells mit der Klasse `XpathAdapter`. Damit wird identifiziert, wie viele Ressourcen welchen Typs erzeugt werden müssen. Viele Konfigurationspunkte des Modells, wie Baudraten oder Basisadressen sind zu vernachlässigen, da sie entweder nicht abgebildet werden können oder nur bei echter Hardware von Bedeutung sind.
Die Kommunikation über *FIFOs*, etwa bei dem UART, erfordert, dass diese vor dem Senden bereits geöffnet wurden. Ist dies nicht der Fall, blockiert der Sendevorgang, was nicht äquivalent mit dem Verhalten echter Hardware ist. Daher erzeugt die Klasse `FifoMaster` die Betriebsmittel und öffnet anschließend die *FIFOs*, über die Geräte nach draußen kommunizieren wollen. *Message Queues* werden von der Klasse `QueueMaster` verwaltet.
Nachdem die Komponenten der Interprozesskommunikation vorbereitet wurden, können die einzelnen SoCs als Kindprozesse durch den `PidMaster` gestartet werden. Die verteilten LavA-Anwendungen müssen vorab für nativen x86-Code übersetzt werden. Über die Kommandozeile werden Daten, wie Identifikationsnummern der Interprozesskommunikation, an die Kindprozesse weitergereicht.
Wenn alle SoCs vernetzt und gestartet wurden, tritt der Simulator in den Hintergrund und wartet mit dem `Listener` auf Ausgaben der einzelnen Kindprozesse. Hierbei ist wichtig, dass die Ausgaben nicht-blockierend gelesen werden, da sonst nur auf eine Quelle gewartet wird, während andere unberücksichtigt bleiben.

4.3 Ressourcenbedarf von MPSoCs

Die prototypische Entwicklung der MPSoCs wird eingeschränkt durch die begrenzten Ressourcen, die ein FPGA zur Verfügung stellt. Daher kann der Fall auftreten, dass die Schaltung eines MPSoCs zu groß ist, im Sinne von benötigten elementaren Bausteinen. Kapitel 4.3.1 definiert den Begriff der Kosten im Kontext von MPSoCs. Anschließend positioniert Kapitel 4.3.2 die Ressourcenabschätzung innerhalb der Werkzeugkette.

4.3.1 Definition der Kosten eines MPSoCs

Um den Begriff der Kosten zu definieren, muss zunächst erläutert werden, wie Schaltungen auf einem *Field-programmable Gate Array*, kurz FPGA, realisiert werden. Die am Lehrstuhl verfügbaren FPGAs stammen vom Hersteller Xilinx [25], sodass im Folgenden dessen Terminologie verwendet wird. Einzelne Strukturen von FPGAs unterschiedlicher Hersteller sind meist ähnlich, jedoch anders benannt.

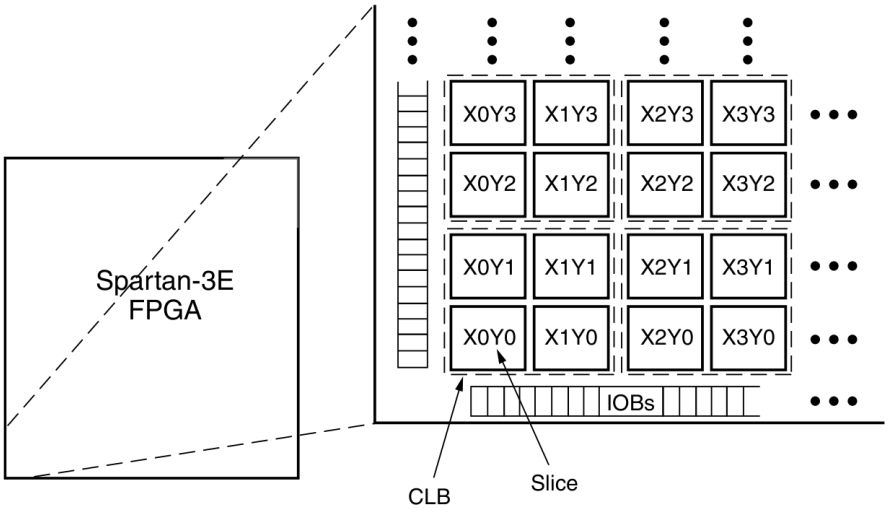

Abbildung 4.2: Aufbau eines Xilinx FPGAs (Quelle: [31])

Ein FPGA besteht aus vielen, gleichmäßig angeordneten, variablen Logikblöcken, den sog. *Configurable Logic Blocks*, kurz CLBs (siehe Abbildung 4.2). Ein CLB ist aufgeteilt in vier *Slices*. Diese enthalten wiederum *Look-Up*-Tabellen, kurz LUTs. Die LUT ist eine programmierbare Wertetabelle mit n Variablen, realisiert durch SRAM-Zellen und Multiplexern. Zusammen mit konfigurierbaren Verschaltungen bilden sie größere Schaltnetze. Ebenfalls in einer *Slice* enthalten sind *Flip-Flops*. Diese stellen einen Speicher für genau ein Bit bereit.
Die Anzahl der LUTs und *Flip-Flops* pro *Slice* variiert je nach Generation des FPGAs: Die *Slice* eines Spartan-3 enthält zwei, die eines Virtex-5 vier der genannten Komponenten. Ebenso hat sich die Zahl der Eingänge einer LUT von vier auf sechs gesteigert. Speicher ist in Form von Block RAMs, welche 2KB bzw. 4KB umfassen, integriert. Auf die Erklärung der übrigen Bausteine wird an dieser Stelle verzichtet.

Ein MPSoC wird also durch diese Strukturen abgebildet. Die Daten und Programme der Prozessoren werden in Block RAMs gespeichert. Der benötigte Speicher dient als erste Kenngröße und muss in Hardware verfügbar sein. Es bleibt eine Einordnung für die Größe der Schaltung zu finden. Da sie keine atomaren Bausteine sind, können einzelne Bestandteile der CLBs sowie *Slices* zu unterschiedlichen Komponenten des MPSoCs gehören. Deshalb sind LUTs die geeignete Einheit, um die Größe einer Schaltung und die Auslastung der Hardware zu beschreiben.
Zusammenfassend bestehen Kosten einer MPSoC-Komponente aus ihrem Bedarf an Block RAMs und LUTs.

4.3.2 Integration in die Werkzeugkette

Zunächst muss ein Kostenmodell aufgestellt werden, mit dem sich die möglichen Varianten der Hardwareplattform bewerten lassen. Ab einer gewissen Komplexität lässt sich aus der beschriebenen Hardware nicht auf später benötigte Ressourcen schließen. Daher müssen umfangreiche Messungen durchgeführt werden, um das Kostenmodell aufzustellen. Besondere Schwierigkeiten ergeben sich hierbei durch die Variantenvielfalt der einzelnen Komponenten sowie deren Komposition.
Die Berechnung des Ressourcenbedarfs folgt direkt, nachdem die Struktur des MPSoCs ermittelt wurde. Diese nahtlose Kopplung ist vorteilhaft, da alle weiteren Arbeiten unnütz sind, falls die gegebene Struktur, auf Grund ihrer Kosten, nicht in Hardware implementiert werden kann. Ohne diesen quantitativen Ansatz würde die Werkzeugkette während der Hardwaresynthese scheitern. Dabei bestimmt die Komplexität des MPSoCs über die Höhe der benötigten Zeit (vgl. Tabelle 6.1).
Die Eingabe der Abschätzung bildet das MPSoC-Modell. Mögliche Implementierungen umfassen eine direkte Integration in den Parser, externe Programme sowie eine modellgetriebene Lösung durch oAW.
Zunächst wurde die Strategie einer Berechnung durch den Parser verfolgt. Kosten wurden abhängig von den *Template*-Parametern der Hardware API berechnet. Offensichtlich ist keine Abschätzung möglich, wenn eine Hardwareklasse keine direkten Parameter besitzt, wie beispielsweise der *Interrupt Controller*. Daher besteht die letztlich implementierte Lösung aus einer regelbasierten Verarbeitung mit oAW.
Unabhängig von der konkreten Implementierung ist der letzte Schritt der Vergleich zwischen den Gesamtkosten und den Kapazitäten des gegebenen FPGAs. Weiter ist denkbar, an dieser Stelle das FPGA auszutauschen, um sich den benötigten Ressourcen anzupassen. Eine Automatisierung wurde hierfür jedoch nicht umgesetzt.

5 Realisierung ausgewählter Werkzeuge

Dieses Kapitel beleuchtet die Implementierung einzelner Aspekte der Werkzeugkette, deren Entwürfe bereits in den Kapiteln 3 und 4 vorgestellt wurden. Kapitel 5.1 zeigt beispielhaft, wie die Arbeit mit der Bibliothek Puma den Parser ermöglichte. Anschließend präsentiert Kapitel 5.2 die Kommunikation zwischen SoCs im Simulator. Die Abschätzung des Ressourcenbedarfs eines MPSoCs erläutert Kapitel 5.3. Danach beschreibt Kapitel 5.4 die Umwandlung des MPSoC-Modells in den SPC der konfigurierbaren Hardwareplattform. Die Automatisierung und Benutzung der Werkzeugkette veranschaulicht Kapitel 5.5.

5.1 Der Parser

Dieses Kapitel betrachtet die Realisierung des Parsers genauer und gibt in Kapitel 5.1.1 einen Überblick über den Umfang der gesamten Implementierung. Anschließend erläutert Kapitel 5.1.2 einen Ausschnitt der Suche nach der Hardware API. Zuletzt beschreibt Kapitel 5.1.3 Probleme, die während der Entwicklung des Parsers auftraten.

5.1.1 Umfang der Implementierung

Der Parser wurde gänzlich in C++ implementiert. Es werden die Bibliotheken Puma und *libxml2* genutzt. Insgesamt erstreckt sich das Programm auf 3438 Zeilen Quelltext und umfasst damit die gesamte Funktionalität, wie sie in Kapitel 4.1 beschrieben wurde.

Parser-Komponente	Programmzeilen	Erklärung
ApiParser	679	Suche der Hardware API Instanzen
InitVisitor	291	Suche der Initialisierungen
HwApiInstance	431	Daten einer Hardware API Instanz
DataContainer	294	Kapselung aller Instanzen eines SoCs
HwConf	535	Manipulation der Anwendung
ModelGenerator	728	Generierung des MPSoC-Modells
sonstiges	480	main, Ausgabeklassen, usw.
Gesamt	3438	

Tabelle 5.1: Umfang der Parser-Komponenten

Tabelle 5.1 präsentiert die Verteilung der Programmzeilen auf die einzelnen Klassen. Der Punkt sonstiges repräsentiert Hilfsklassen, z.B. für die Ausgabe von Informationen auf der Konsole.
Eine genauere Einteilung der Programmzeilen auf die Kerngebiete des Parsers veranschaulicht Abbildung 5.1. Die Suche nach Instanziierungen und Initialisierungen umfasst die Klassen `ApiParser` und `InitVisitor`. Die Datenhaltung besteht aus `DataContainer` und `HwApiInstance`. Die Manipulation wird einzig in `HwConf` realisiert, während die Modellgenerierung in `ModelGenerator` platziert ist.

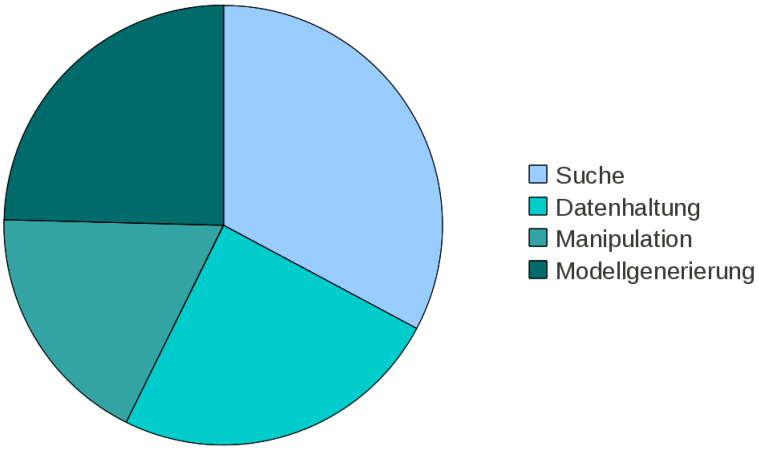

Abbildung 5.1: Verteilung der Programmzeilen auf die Kernbereiche des Parsers

5.1.2 Suche nach globalen Objekten

Vor jeder Analyse muss PUMA einen Quelltext scannen und parsen. Damit werden erst die einzelnen Zeichen zu Symbolen zusammengefügt und anschließend in dem Syntaxbaum und der semantischen Datenbank organisiert. Quelltext 5.1 zeigt, wie man von einem Dateinamen zur semantischen Datenbank gelangt.

```
// Objekte für die Ausgabe und das Projekt anlegen
ErrorStream err;
CProject project = new CProject(err);

// Die Übersetzungseinheit scannen
Unit *unit = project.scanFile(filename);

// Die Übersetzungseinheit parsen
CCParser parser;
CTranslationUnit *tu = parser.parse(*unit, project);

// Die semantische Datenbank
CSemDatabase &database = tu->db();
```

Quelltext 5.1: Semantische Analyse einer Quelltextdatei

Mit Hilfe der Datenbank lassen sich alle semantischen Informationen komfortabel durchsuchen. Die Hardware API kann in statischen Klassenattributen, globalen Variablen und Singleton-Methoden instanziiert werden. Hier sollen nur die beiden ersten Ansätze verfolgt werden. Statische Klassenattribute und globale Variablen bildet PUMA durch Objekte der Klasse `CAttributeInfo` ab. Der folgende Quelltext 5.2 iteriert über alle semantischen Objekte und untersucht diese Attribute intensiver mit der Methode `processAttributeInfo`.

```
// Über alle semantischen Informationen iterieren
CSemDatabase::ObjectSet& objects = database.Objects();
CSemDatabase::ObjectSet::iterator objIter = objects.begin();

for (; objIter != objects.end(); objIter++) {
  // Prüfen, ob semantisches Objekt ein Attribut ist
  if ((*objIter)->AttributeInfo()) {
    // Auf Hardware API untersuchen
    processAttributeInfo(objInfo->AttributeInfo());
  }
}
```

Quelltext 5.2: Iteration über alle semantischen Informationen

Die Methode `processAttributeInfo` in Quelltext 5.3 grenzt erstmals die Attribute ein, welche eine Hardware API Instanz enthalten können. Der Typ eines Objekts muss auf eine Klasse hindeuten und schließt damit einfache Datentypen aus. Felder von Objekten sind ebenfalls erlaubt. Die nächste Abfrage prüft, ob das Objekt tatsächlich als globales oder als statisches Klassenattribut deklariert wurde. Die Methode wird verlassen, falls eine Bedingung nicht erfüllt wird. Andernfalls startet der in Kapitel 4.1.1.2 beschriebene Algorithmus ab Zeile 12 mit dem hier aufgeführten Objekt `attr` als Eingabe.

5.1.3 Schwierigkeiten bei der Entwicklung

Der Umgang mit C++ *Templates* ist sehr kompliziert und fehleranfällig. Dieses Sprachkonstrukt ermöglicht Metaprogrammierung [24, 23], die zur Übersetzungszeit geschieht. Natürlich wird dies erst durch entsprechende Parser und Compiler bewerkstelligt, die mit der Instanziierung und Spezialisierung von *Templates* vertraut sind.

Leider kam es bei der Entwicklung mit PUMA zu einigen Fehlern in diesem Bereich. Wenn der Parser einen Fehler bei der Verarbeitung der Hardware API signalisierte, war unklar, ob der Fehler im Parser, der Hardware API oder bei PUMA lag. Falls anschließend der *GNU Compiler* die gleiche Eingabe fehlerfrei übersetzte, war der Fehler meist bei PUMA zu suchen.

Glücklicherweise war eine direkte Verbindung mit den Autoren von PUMA möglich, sodass vorgefundene Fehler gemeldet, diskutiert und zeitnah behoben werden konnten. Die Verarbeitung von *Templates* durch PUMA ist daher fehlerfreier geworden und genügt den, durch diese Diplomarbeit gestellten, Ansprüchen.

```
void ApiParser::processAttributeInfo(CAttributeInfo *attr) {
  // Attribut muss Objekt einer Klasse sein oder ein Feld
  int arrayBounds = 0;
  CTypeInfo *typeInfo = attr->TypeInfo();
  if (!typeInfo || !typeInfo->isClass()) {
    if (typeInfo->isArray() && typeInfo->BaseType()
                      && typeInfo->BaseType()->isClass()) {
      arrayBounds = typeInfo->Dimension();
    } else {
      return;
    }
  }

  // Nur statische Klassenattribute oder globale Variablen zulassen
  CScopeInfo *scope = attr->Scope();
  if (!scope || !(scope->GlobalScope() || scope->isNamespace() ||
      (attr->isClassMember() && attr->isStatic()))) {
    return;
  }

  processSingleObject(attr, arrayBounds);
}
```

Quelltext 5.3: Erste Überprüfung der Objekte

5.2 Der Simulator

Hier wird die Umsetzung der MPSoC-Simulation erläutert. Dazu stellt Kapitel 5.2.1 den Gesamtaufwand der Implementierung vor. Danach veranschaulicht Kapitel 5.2.2 die Kommunikation zwischen den simulierten SoCs.

5.2.1 Umfang der Implementierung

Der LavA-Simulator wurde mit insgesamt sechs Klassen modular implementiert und verteilt sich auf 1123 Zeilen Programmcode. Die genaue Zuordnung präsentiert Tabelle 5.2. Zurzeit sind lediglich die Peripheriegeräte IPC und UART im Simulator implementiert, wovon die Realisierung des IPC Controllers in Kapitel 5.2.2 genauer betrachtet wird.

5.2.2 Die Umsetzung der IPC Controller

Die Kommunikation ist ein Hauptbestandteil jedes MPSoCs. In Abbildung 5.2 wird die Vernetzung auf der Hardware und per Simulator gegenübergestellt. Auf Hardwareebene verschaltet der IPC-Bus die einzelnen SoCs miteinander. In der Simulation gibt es eigene, unabhängige Prozesse, die durch *Message Queues* und den Simulator miteinander verbunden werden.

Simulator-Komponente	Programmzeilen	Erklärung
XpathAdapter	179	Auswerten des MPSoC-Modells
DescriptorContainer	51	Kapselt Deskriptoren
FifoMaster	227	Behandlung der FIFOs
QueueMaster	276	Behandlung der Message Queues
PidMaster	191	Verwaltung der Kindprozesse
Listener	138	Verarbeitung aller Eingaben
main	61	Einsprungpunkt
Gesamt	1123	

Tabelle 5.2: Umfang der Simulator-Komponenten

Der IPC Controller aus der Hardware API musste neu implementiert werden, um die Anbindung an die gezeigte Kommunikation zu gewährleisten. Nachfolgend wird in Kapitel 5.2.2.1 die Klasse `QueueMaster` des Simulators beschrieben, welche die *Message Queues* erzeugt und die Nachrichten weiterleitet. Anschließend zeigt Kapitel 5.2.2.2 exemplarisch, wie die Funktionalität der Hardware API auf die Simulation zugeschnitten wird.

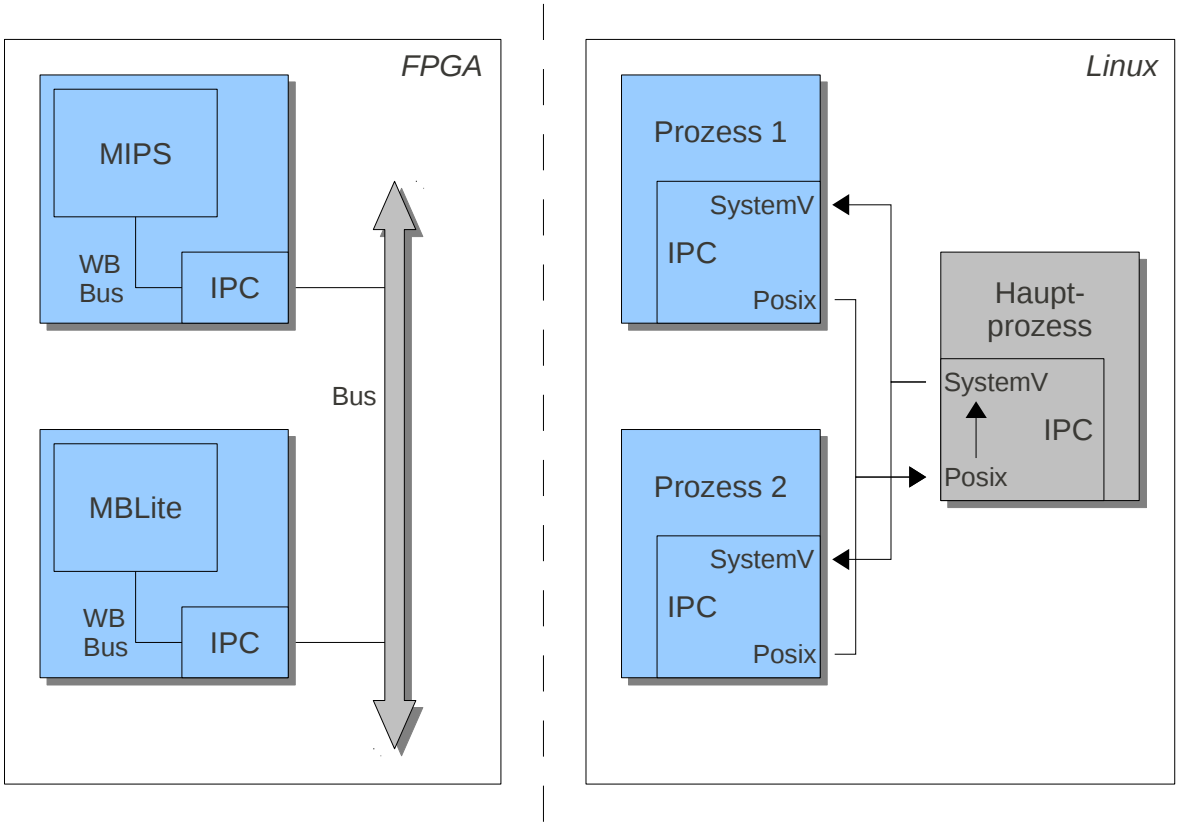

Abbildung 5.2: Vergleich der Kommunikation auf Hardware und per Simulation

5.2.2.1 Initialisieren der Message Queues im Simulator

Der Simulator überprüft zunächst das MPSoC-Modell auf vorhandene Kommunikationsstrukturen. In Quelltext 5.4 ist der hier relevante Teil für obiges Modell dargestellt.

```
<ConnList SoCList="//@SoCList[Id='1'] //@SoCList[Id='2']" Id="3"/>
<SoCList Id="1">
  <_Devices xsi:type="IPC" _Base="2281703424" _UCF="0"
                           Bus="//@ConnList[Id='3']"/>
</SoCList>
<SoCList Id="2">
  <_Devices xsi:type="IPC" _Base="2281703424" _UCF="0"
                           Bus="//@ConnList[Id='3']"/>
</SoCList>
```

Quelltext 5.4: MPSoC-Teilmodell für Kommunikation zwischen zwei SoCs

Für jede Verbindung werden eine *POSIX* und *System V Message Queue* angelegt. Dieser Ansatz ist durch die Emfangsbedingungen der beiden Warteschlangen begründet. Der Simulator behandelt Eingaben über ein Multiplexing mit der Funktion `select`. Diese erhält eine Menge von Dateideskriptoren und prüft gleichzeitig, ob und welcher Deskriptor Daten bereithält. Die *POSIX Queue* unterstützt solche Deskriptoren. Die SoC-Prozesse hingegen empfangen nur Nachrichten einer bestimmten Identifikationsnummer, ihrer SoC-Nummer. Dieses Verhalten lässt sich durch *System V Queues* abbilden.
Zunächst gilt es, die Identifikationsnummer der Verbindungen zu extrahieren. Dies wird in der Klasse `XpathAdapter` gelöst. Danach erzeugt der Simulator mit `mq_open` die *POSIX Message Queue*. Der Name der Warteschlange wird zusammengesetzt aus „/lava_mq_“ und der entsprechenden Verbindungsnummer als Suffix. Als Resultat wird der Dateideskriptor gespeichert. Nun erstellt `msgget` die *System V Message Queue*. Hier ist eine Schlüsselnummer gefordert. Der Simulator beginnt mit einem festen Wert, den er bei Misserfolg schrittweise inkrementiert. Bei Erfolg gibt der Linux-Kern eine eindeutige Nummer zurück.
Eine Zuordnung von dem Dateideskriptor der *POSIX*-Warteschlange zur Identifikationsnummer der *System V*-Verbindung sichert die Weiterleitung der empfangenen Daten an den richtigen Sendekanal. Die empfangenen Botschaften werden zusätzlich ausgegeben und geben damit einen Überblick über die internen Abläufe des MPSoCs.

5.2.2.2 Alternative Implementierung des IPC Controllers

Der IPC Controller in den SoC-Prozessen stellt die zweite Komponente der Kommunikationsverbindungen dar. Die ursprünglichen Speicherzugriffe werden nun durch Systemaufrufe ersetzt, um über den Simulator miteinander zu kommunizieren.
Zuerst wird die Peripherie der Hardware API mit der *Template*-Methode `instance` initialisiert. Tabelle 5.3 zeigt eine Kommandozeile, wie sie vom Simulator zusammengesetzt wird. Die passenden Werte werden über den *Template*-Parameter adressiert. Das Suffix für den Namen der *POSIX Queue* befindet sich für IPC Controller i in Parameter $2*i+2$. Der Schlüssel für die *System V Queue* ist in Parameter $2*i+3$. Somit ist sichergestellt, dass IPC Controller und Verbindung zur Laufzeit korrekt verbunden werden.

Index	0	1	2	3	4	5
Wert	./1.elf	1	3	305397760	5	305397762
Erläuterung	Aufruf	SoC ID	*POSIX* Suffix	*System V* Schlüssel	*POSIX* Suffix	*System V* Schlüssel
			Werte für IPC Controller 0		Werte für IPC Controller 1	

Tabelle 5.3: Kommandozeile (grau) für einen SoC-Prozess

In Quelltext 5.2.2.2 wird der simulierte Schreibzugriff präsentiert. Neben den eigentlichen Daten, wird ein *Offset* zur Speicheradresse des Controllers angegeben. Da kein Speicherzugriff stattfindet, wird mit dem *Offset* die nächste Aktion ausgewählt. Hier können entweder die Zieladresse der Kommunikation, oder die Daten der Botschaft übergeben werden. Letztere werden sofort an die vorher gespeicherte Adresse verschickt. Dazu wird eine Nachricht, bestehend aus Start- und Zieladresse sowie Daten, erstellt und mit `mq_send` an die *POSIX Message Queue* weitergereicht.

```
volatile void write32(unsigned int offset, long data) {
  // Abfrage nach spezifiziertem Register
  if (offset == IPCADDR) {
    // Setzen der Adresse löst keine Kommunikation aus
    writeAddr = data;
  } else if (offset == IPCDATA) {
    // Daten werden sofort an gespeicherte Adresse geschickt!

    // Paket füllen
    MsgType msg;
    msg.target = writeAddr;
    msg.source = mySocId;
    msg.data[3] = (char) ((data >> 24) & 0xFF);
    msg.data[2] = (char) ((data >> 16) & 0xFF);
    msg.data[1] = (char) ((data >> 8) & 0xFF);
    msg.data[0] = (char) (data & 0xFF);

    // Paket abschicken über die POSIX-Queue
    if (mq_send(sendId, (char *) &msg, sizeof(MsgType), 1) == -1) {
      // Fehlerbehandlung
    }
  }
}
```

Quelltext 5.5: Schreibzugriff auf den simulierten IPC Controller

5.3 Berechnung des Ressourcenbedarfs

Für eine effektive Berechnung des Ressourcenverbrauchs muss ein Modell aufgestellt werden. In Kapitel 5.3.1 werden die durchgeführten Messungen sowie das resultierende Modell beschrieben. Danach erläutert Kapitel 5.3.2, wie die Kosten im Metamodell verankert werden. Kapitel 5.3.3 veranschaulicht die implementierte Berechnung für ein gegebenes MPSoC.

5.3.1 Das Ressourcenmodell

Die Kosten einer Schaltung sind in Kapitel 4.3.1 durch den Bedarf von LUTs und Block RAMs definiert. Weiter steht fest, dass zahlreiche Messungen alle Varianten der Hardwareplattform abdecken müssen, um ein Ressourcenmodell aufstellen zu können. Im Folgenden wird eine teilautomatische Lösung für dieses Problem gezeigt.
Zunächst muss geklärt werden, wie man die nötigen Messergebnisse erhält. Xilinx' Werkzeugkette stellt dazu Angaben in dem *MAP Report File* (Dateiendung „mrp“) zur Verfügung. Der Bericht beginnt mit einer Übersicht über den Gesamtverbrauch an Ressourcen durch das gegenwärtige Design. Dies ist jedoch zu grobgranular und erlaubt keine genaue Zuordnung. Seit Version 10 der Entwicklungsumgebung *ISE* ist eine hierarchisch aufgeschlüsselte Verteilung der Kosten in Bereich 14 des o.g. Berichts zu finden.

```
+------------------------------------------------------------
| Module                       | Slices    | LUTs       | BRAM   ...
+------------------------------------------------------------
| mpsoc_top/                   | 674/15465 | 397/19985  | 0/22
| +soc_top_1                   | 42/3883   | 50/5138    | 0/6
| ++can_inst_1                 | 1/833     | 1/1061     | 0/2
| +++inst_can_controller       | 19/832    | 16/1060    | 0/2
| ++++Inst_can_bus_interfac    | 146/146   | 161/161    | 0/0
...
| ++cpu                        | 36/2439   | 36/3416    | 0/4
...
| ++interrupt_inst             | 5/5       | 5/5        | 0/0
...
```

Quelltext 5.6: Auszug aus dem „MAP Report File“

Quelltext 5.6 zeigt den Verbrauch von *Slices*, LUTs und Block RAMs für die Komponenten MPSoC, SoC1, CAN, CPU1 und den Interrupt Controller. In den Zeilen sind die *Entities*, wie sie in der Hardwarebeschreibungssprache definiert werden, aufgelistet. Durch die Einrückung mit dem +-Symbol wird die Hierarchie und damit eine „enthalten-in“-Beziehung hergestellt. Alle Komponenten gehören demnach zum MPSoC, gliedern sich dann in SoCs usw.
Die Kosten sind nach Spalten sortiert. Jeder Eintrag enthält zwei Teile: Vor dem Schrägstrich sind die Kosten aufgeführt, die durch die *Entity* in der jeweiligen Zeile entstehen. Anschließend folgen die summierten Kosten durch die aktuelle sowie aller untergeordneten Komponenten.
Aus dem Beispiel lässt sich ablesen, dass die CPU einen Verbrauch von insgesamt 3416 LUTs und vier Block RAMs hat. Der Interrupt Controller hingegen benötigt lediglich fünf LUTs. Diese Kosten variieren: Falls die CPU mehr Speicher zugewiesen bekommt, wird die zusätzliche Ansteuerung den Bedarf anheben. Ebenso dominiert die Anzahl der Geräte mit verfügbarer Unterbrechung die Kosten für den eingesetzten Interrupt Controller.
Die Anzahl der möglichen Unterbrechungen ist momentan auf 32 beschränkt. Um den Ressourcenverbrauch des Interrupt Controllers exakt zu vermessen, wären demnach 32 synthetisierte Varianten notwendig. Genauso gut können wenige Messungen ausreichen, um Tendenzen sichtbar zu machen, wie in Abbildung 5.3.

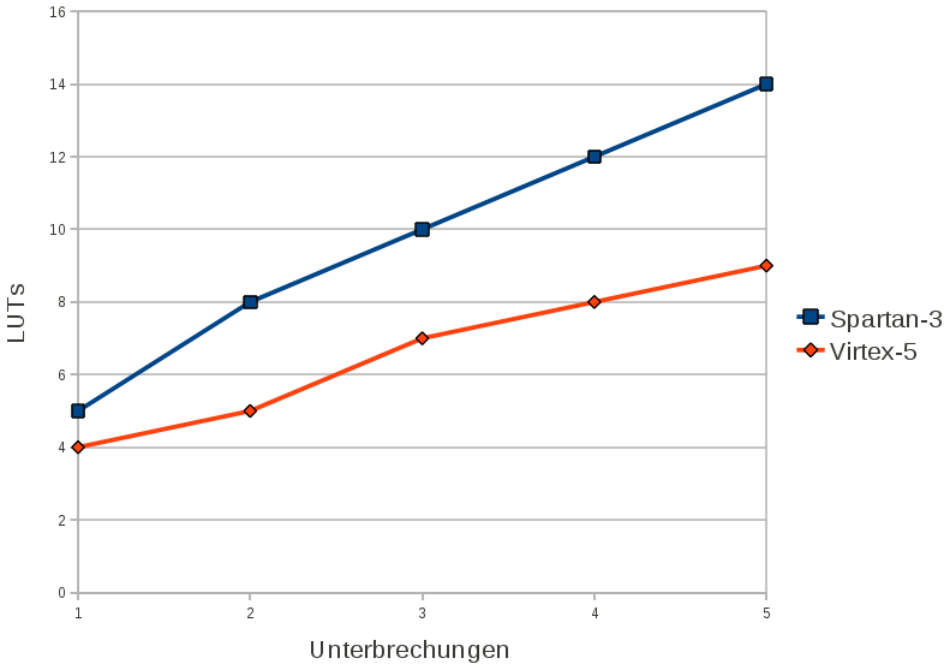

Abbildung 5.3: LUT-Verbrauch des Interrupt Controllers

Ein anderes Beispiel ist der CAN Controller. Dieser kann über zwei boolesche Variablen für die Nutzung von *Filter* und *Buffer* konfiguriert werden. Vier Messungen ergaben die in Quelltext 5.7 dargestellten Kostenfunktionen.

```
Spartan-3
costs_LUTs(Filter, Buffer) = 933 + Filter * 47 + Buffer * 90
costs_BRAM(Filter, Buffer) = Filter + Buffer

Virtex-5
costs_LUTs(Filter, Buffer) = 760 + Filter * 41 + Buffer * 154
costs_BRAM(Filter, Buffer) = Filter + Buffer
```

Quelltext 5.7: LUT- und Block RAM-Verbrauch des CAN Controllers

Aus obigen Beispielen und weiteren Messungen resultiert, dass das Ressourcenmodell linear sein muss. Die einzelnen Komponenten wachsen somit proportional zu ihrer Konfiguration. Problematisch sind jedoch Strukturen wie SoC oder MPSoC, die wenig Logik besitzen und stattdessen andere Bausteine verschalten. Auf Grund der hohen Vielfalt an Komponenten, die zur Auswahl stehen, konnten bis dato keine aussagekräftigen Kostenfunktionen aufgestellt werden. Stattdessen wurden simple Näherungsfunktionen genutzt, damit deren Kosten nicht gänzlich unberücksichtigt bleiben. Diese Funktionen sind proportional abhängig von der Anzahl der zu integrierenden Bestandteile.
Systematische Messungen lassen sich mit vorher angefertigten Konfigurationsdateien für XVCL realisieren. Eine Synthese wird unter Linux mit nur einem Thread ausgeführt, weshalb mehrere parallele Synthesen nötig sind, um die Arbeit zu beschleunigen. Vor dem

Prozess *Place & Route* wird Xilinx' Werkzeugkette gestoppt, da der o.g. Bericht bereits vorher erzeugt wurde. Dies spart zusätzliche Zeit bei den Messungen ein. Anschließend müssen die Ergebnisse manuell ausgewertet werden.

5.3.2 Modellierung der Kosten in der Hardware API

Wie alle Informationen zur Hardware API, werden auch die Kostenfunktionen im Metamodell (siehe Kapitel 3.2.1.1) gespeichert. Attribute mit den Namen `_C_LUT4`, `_C_LUT6`, `_C_BRAM2` und `_C_BRAM4` spezifizieren die Kosten für den Bedarf an LUTs sowie Block RAMs. Die nachgestellten Ziffern unterscheiden die jeweilige Variante, die, wie in Kapitel 4.3.1 beschrieben, vom verwendeten FPGA abhängt. Kapazitäten der FPGAs werden in ihren Klassen definiert. Dabei werden ebenfalls obige Attribute genutzt, wobei nur eine Variante für LUT und Block RAM sinnvoll ist.
Da die Berechnung der Kosten in oAW mit der Sprache *Xtend* realisiert wird, müssen sie in der oAW-spezifischen Syntax angegeben werden. Die Kostenfunktionen werden in dem Feld `defaultValueLiteral` eingetragen. Es können feste Größen oder Formeln beschrieben werden, die abhängig von den jeweiligen Attributen und Referenzen der Klasse sind. Das vorherige Beispiel des CAN Controllers schildert eine solche Formel, die in Quelltext 5.8 veranschaulicht wird.

```
Integer getCosts(CAN can, SoC _soc, String type) :
  switch (type) {
    case "_C_LUT4" : (933 + (can.Filter ? 1 : 0) * 47
                          + (can.Buffer ? 1 : 0) * 90)
    default : 0
  }
;

Integer getCosts(IRQController irq, SoC _soc, String type) :
  switch (type) {
    case "_C_LUT4" : (2.2 * _soc._Devices._IRQ.select(e|e>=0).size + 3)
    default : 0
  }
;
```

Quelltext 5.8: Abbildung der Kostenfunktionen in *Xtend*

Der obige Quelltext wurde automatisiert aus dem Metamodell generiert. Die abgebildeten Funktionen geben einen Überblick über die mögliche Modellierung der Kosten. Diese können neben den Attributen und Referenzen der eigenen Klasse zusätzlich auf die Daten des aktuellen SoCs zurückgreifen, wie die zweite Regel veranschaulicht.

5.3.3 Abschätzung der Kosten

Die Kostenberechnung ist eingebettet in der Validierung des erzeugten MPSoC-Modells. Der in Quelltext 5.9 enthaltene Aufruf `checkCosts` leitet die Berechnung an ein Gerüst von *Xtend*-Funktionen weiter. Die Regel wird ignoriert, wenn kein FPGA im Modell enthalten ist. Dies tritt vor allem bei der Simulation auf.

```
// checks.chk
context MPSoC if (FPGA != null) ERROR
  "The FPGA is too small for the design!" :
  checkCosts();

// calculator.ext
Boolean checkCosts(MPSoC mpsoc) :
  let list = getCostNames(mpsoc.FPGA) :
  list.forAll(e|mpsoc.FPGA.getCapacity(e) >=
                  mpsoc.SoCList.getSocCosts(e).summarize())
;
```

Quelltext 5.9: Kostenberechnung als *Check*-Bedingung

Es werden zunächst die Kostentypen für das aktuelle FPGA bestimmt. Diese werden durch eine Liste von Zeichenketten definiert. Anschließend erfolgt der Vergleich von Kapazität und Verbrauch, der für jeden Kostentyp positiv ausfallen muss.
Im Prinzip werden die Gesamtkosten entlang einer Baumstruktur berechnet, wobei jeder Knoten eigene Kosten definieren kann und daher besucht werden muss. Das MPSoC bildet den Wurzelknoten und trägt dadurch die Gesamtkosten. Die Peripherie bildet die Blattknoten, die keine weiteren Unterfunktionen aufrufen, sondern lediglich eine Kostenfunktion, wie in Quelltext 5.8, enthalten. Die Traversierung und Berechnung der Kosten kann daher in einer Datei abgelegt werden, während die eigentlichen Kostenfunktionen der Blattknoten mit einer zusätzlichen Datei generiert werden.

5.4 Vom MPSoC-Modell zur Hardwarekonfigurierung

Der Parser speichert die Struktur des MPSoCs in einem XML-Modell ab. Die konfigurierbare Hardwareplattform benötigt jedoch eine anders formatierte Konfigurationsdatei, den SPC. Diese Lücke schließt ein zweites oAW-Projekt, welches die Modell-zu-Modell-Transformation durchführt. Hinzu kommen einige Abfragen, die das Modell vorab semantisch überprüfen, damit eine gültige Hardwarekonfiguration erstellt werden kann. Damit ist ebenfalls die Kostenabschätzung des vorherigen Kapitels realisiert worden.

Die Implementierung der Abfragen wird in Kapitel 5.4.1 präsentiert. Anschließend erläutert Kapitel 5.4.2, wie der SPC erzeugt und damit die Verknüpfung zur restlichen Werkzeugkette hergestellt wird.

5.4.1 Validierung des MPSoC-Modells

Der Parser sorgt während der Modellgenerierung für eine korrekte Syntax, wodurch oAW erst in der Lage ist, das Modell überhaupt zu öffnen. Syntaktische Korrektheit genügt an dieser Stelle jedoch nicht. Das Metamodell aus Kapitel 3.2.1.1 stellt, durch Attribute und Referenzen, nur grobe Regeln auf, die hier verfeinert werden müssen.

Wie bereits bei der Generierung der Hardware API, löst diese Aufgabe eine Validierung mit statischen Bedingungen. Definiert werden sie durch die einzelnen Konfigurationspunkte der Hardwareplattform. Es folgen ein paar Beispiele.

- Falls ein Gerät Unterbrechungen nutzt, ist ein Interrupt Controller im SoC vorhanden.
- Jedes SoC benötigt mindestens eine Peripherie.
- Es dürfen keine zwei Interrupt Controller in einem SoC auftreten.
- Kommunikationsverbindungen müssen 8, 16, 24 oder 32Bit breit sein.
- Geräte mit Außenanbindung haben unterschiedliche Instanznummern (und damit auch Pins).

Exemplarisch soll die erste Regel hier näher ausgeführt werden. Quelltext 5.10 demonstriert sie in der Sprache *Check*. Wie zu sehen, bezieht sich die Regel nur auf Knoten vom Typ SoC. Eine weitere Einschränkung begrenzt dies auf SoCs, die mindestens ein Gerät enthalten. Im Falle des Misserfolges wird der Bedarf eines Interrupt Controllers für das SoC mit angegebener Identifikationsnummer ausgegeben. Die Bedingung wird erfüllt, wenn entweder alle Geräte auf Unterbrechungen verzichten, oder ein Gerät vom Typ Interrupt Controller vorhanden ist.

```
// Interrupt Controller vorhanden, wenn Unterbrechungen benötigt werden
context SoC if (_Devices.size > 0) ERROR
  "SoC (ID " + Id + ") needs an IRQ-Controller" :
  (_Devices.forAll(e|e._IRQ == -1)) ||
  (_Devices.exists(e|e.metaType == LavA::IRQController));
```

Quelltext 5.10: Überprüfung der SoCs auf benötigte Interrupt Controller

5.4.2 Generierung des SPCs

Die automatisierte Ableitung des SPCs wird mit *Xpand* gelöst. In diesem Prozess müssen alle im Modell enthaltenen Informationen in die Konfigurationsdatei eingehen. Das *Template* kapselt den jeweils aktuellen SPC. Aktualisierungen und Änderungen der Hardwareplattform müssen daher unbedingt hier eingepflegt werden, um Versionskonflikte bei der Hardwaregenerierung zu vermeiden.
Der obige Quelltext zeigt, wie die Variablen für die Kommunikationsverbindungen gesetzt werden. Diese umfassen Namen, Identifikationsnummer, Typ, Busbreite und die Größen der ein- sowie ausgehenden Puffer. Ihre Werte stammen aus den `Connection`-Knoten, die im MPSoC unter der Referenz `ConnList` gespeichert werden. Von diesen Knoten werden die einzelnen Attribute zu einer Liste zusammengefügt. Anschließend wird über eine `EXPAND`-Anweisung die Funktion `printVar` aufgerufen, welche die Variablen und Werte durch einen XML-Knoten erzeugt.

```
« EXPAND printVar("IPC", ConnList.collect(e|e.Type.toString()
                                          + e.Id.toString())) »
« EXPAND printVar("IPCIDs", ConnList.collect(e|e.Id)) »
« EXPAND printVar("IPCTyp", ConnList.collect(e|e.Type)) »
« EXPAND printVar("IPCBusBreite", ConnList.collect(e|e.Width)) »
« EXPAND printVar("MsgQueue", ConnList.collect(e|e.MessageQueue)) »
« EXPAND printVar("OutBuffer", ConnList.collect(e|e.OutBuffer)) »
  ...

« DEFINE printVar(String name, List values) FOR Object »
  <set-multi var=' « name » ' value=' « values.toString() » '/>
« ENDDEFINE »
```

Quelltext 5.11: Übertragen der Werte aus dem MPSoC-Modell in den SPC

5.5 Automatisierung der Werkzeugkette

Implementiert wurde die Werkzeugkette durch das Skript „lava.sh“ für die *GNU Bourne-again shell* [32], kurz Bash, und fasst die in Kapitel 3.3 genannten Schritte zusammen.

```
$ ./lava.sh -h
usage: ./lava.sh [-nc] [-src=<dir>] [-list] <target>

-nc        Do not compile the code
-src=*     The directory containing the MPSoC-application
-list      List the currently supported FPGAs
<target>   Specify the FPGA or use the string "sim" for simulation
```

Quelltext 5.12: Die Verwendung von „lava.sh“

Die Parameter werden nachfolgend erläutert:

-nc Verhindert, dass die Werkzeugkette die LavA-Anwendung kompiliert. Da auch das Weben von Aspekten verhindert wird, muss dies vorher manuell geschehen sein, damit der Parser gewobene Quelltextdateien vorfindet.

-src Mit diesem Parameter legt man den Quellordner fest, in dem sich die verteilte LavA-Anwendung befindet. Der Ordner muss pro SoC ein Unterverzeichnis aufweisen. Standardmäßig ist der Quellordner auf „<Working Directory>/parse/mpsoc“ gesetzt.

-list Die aktuell unterstützten FPGAs können mit diesem Parameter angezeigt werden.

<target> Die Zielarchitektur wird zuletzt angegeben und besteht entweder aus einem Eintrag der Liste des vorigen Parameters oder dem Ausdruck „sim“. Letzteres verhindert die Synthese und startet stattdessen den Simulator. Die Anwendung muss für die Simulation entsprechend als x86-Programm übersetzt werden!

Eingebettet ist das Skript in eine Ordnerstruktur, die alle Teilprogramme enthält. Über `export`-Anweisungen werden Variablen, z.B. die Zielarchitektur, für weitere Skripte bekannt gemacht, um dort zusätzliche Parameter zu vermeiden. In Quelltext 5.13 ist die Ordnerstruktur der Werkzeugkette aufgelistet.

```
root/                Oberster Ordner der Werkzeugkette; enthält lava.sh
root/parse/          Enthält den Parser sowie dessen generiertes Modell
root/parse/mpsoc/    Quellverzeichnis der LavA-Anwendung
root/oaw/            Das Projekt, welches den SPC erzeugt
root/xvcl/           XVCL und die konfigurierbare Hardwareplattform
root/vhdl/           Arbeitsordner für die Synthese der Hardware
root/final/          Einsetzen der Anwendung in das Bitfile
root/sim/            Besitzt Symlink und Skript zur Simulation des MPSoC
```

Quelltext 5.13: Die Ordnerstruktur der Werkzeugkette

Die Zwischenergebnisse werden über *Symlinks* verteilt: Der Parser generiert das Modell im Ordner „root/parse/“, welches über einen relativen Link gleichzeitig in „root/oaw/“ vorhanden ist. Mit diesem Ansatz werden auch Programme oder Teilprojekte, wie der Parser selbst, eingebunden, wenn sie an anderer Stelle entwickelt werden.

6 Evaluation der entwickelten Werkzeugkette

Nachdem die Werkzeugkette fertiggestellt wurde, sollen in diesem Kapitel ihre Anwendungsmöglichkeiten untersucht werden. Zunächst präsentiert Kapitel 6.1 die durchgängige Konfigurierung von LavA-Anwendungen. Anschließend veranschaulicht Kapitel 6.2 die Arbeit mit dem Simulator und diskutiert die Unterschiede zur Entwicklung mit echter Hardware. Danach werden in Kapitel 6.3 die Ergebnisse der Kostenabschätzung betrachtet.

6.1 Durchgängie Konfigurierung einer CAN Anwendung

Dieses Kapitel stellt eine konfigurierbare LavA-Anwendung vor, die eine variable Anzahl an CAN-Bussen vernetzt. Es wird die durchgängige Konfigurierung von Hard- und Software verdeutlicht. In Kapitel 6.1.1 wird erst die Anwendung genauer vorgestellt. Darauf folgt in Kapitel 6.1.2 die Implementierung der Applikation. Abschließend wird in Kapitel 6.1.3 die Konfigurierung der Software und die dadurch implizierte Maßschneiderung der Hardware gezeigt.

6.1.1 Die Anwendung

Die Anwendung ist unterteilt in mehrere *Threads* des Betriebssystems CiAO [3], welches in [26] speziell auf LavA zugeschnitten wurde. Die Aufgaben der *Threads* werden definiert durch die Behandlung der folgenden Hardware:

UART Der UART-*Thread* dient als zentrale Ein- und Ausgabeschnittstelle des Systems. Er kann Befehle von außerhalb empfangen und damit die anderen *Threads* steuern, oder Daten zu Diagnosezwecken liefern.

CAN Jeder CAN Controller wird mit einem externen CAN-Bus vernetzt. Primär sollen CAN-Botschaften empfangen werden. Die Architektur erlaubt jedoch Nachrichten abzusenden, etwa auf Befehl des UART-*Threads*.

Die Anzahl der *Threads* ist für den CAN-*Thread* variabel, für den UART-*Thread* jedoch mit maximal einer Instanz bemessen. Durch die Anwendung sollen zwei Einsatzgebiete abgedeckt werden: Die Diagnose einer variablen Anzahl von CAN-Netzwerken sowie der Aufbau eines *Gateways* zwischen diesen. Der UART ist für das *Gateway* nicht zwingend notwendig.

6.1.2 Implementierung der Threads

In Quelltext 6.1 wird das Schema des UART-*Threads* beschrieben. Die Methode `functionTaskUART` der Anwendung `LavA` enthält die Funktionalität. Diese wartet auf Eingaben durch einen angeschlossenen *Terminal* und verarbeitet eintreffende Kommandos. Befehle an andere *Threads* werden über den aufgeführten IPC Controller verschickt.

```
void LavA::functionTaskUART() {
  hw::dev::IPC0 &ipc0 = hw::dev::IPC0::Inst();
  while (1) {
    char c = hw::dev::UART0::Inst().getc();
    /* verarbeite Kommandos */
  }
}
```

Quelltext 6.1: Implementierung des UART-*Threads*

Im vorigen Kapitel wurden zwei Einsatzgebiete aufgezeigt, die Diagnose und das *Gateway*. Natürlich spiegelt sich die Entscheidung für eine dieser Anwendungen in der Implementierung des CAN-*Threads* wieder. Die Diagnose des CAN-Bus 0 veranschaulicht Quelltext 6.2. Empfangene CAN-Botschaften werden über den IPC-Bus an den UART-*Thread* geschickt, welcher die Daten an die Konsole weiterleiten kann. Die *Gateway*-Funktionalität wird per internem *Broadcast* der empfangenen CAN-Botschaft bewerkstelligt.

```
void LavA::functionTaskCAN0() {
  while (1) {
    struct can_message msg;
    hw::dev::CAN0::Inst().receive(msg);
    hw::dev::IPC0::Inst().send(0, msg.id);
    hw::dev::IPC0::Inst().send(0, msg.data);
  }
}
```

Quelltext 6.2: Diagnose des CAN-Bus

Die genutzten Klassen und Namensräume werden durch das Betriebssystem bereitgestellt. Eine Applikation für CiAO wird üblicherweise durch eine Klasse abgebildet. Tasks werden mit Methoden realisiert. Neben anderen Einstellungen, wird die Definition von Tasks in der Konfigurationsdatei „config.xml“ vorgenommen. Ein Auszug dieser Datei wird in Quelltext 6.3 gezeigt.

6.1.3 Durchgängige Konfigurierung von Hard- und Software

Die erste Konfigurierung bei Applikationen für CiAO findet durch das Variantenmanagementprogramm *pure::variants* [33] statt. Dieses Programm ermöglicht die strukturierte Abbildung variantenreicher Lösungen eines Problems. Durch eine Auswahl von *Features* werden der speziellen Lösung Artefakte hinzugefügt. Bezogen auf das aktuelle Beispiel wird CiAO so konfiguriert, dass die gewünschte Peripherie (maximal ein UART, mehrere CAN Controller) verfügbar ist.

```
<?xml version="1.0"?>
<ciaoApp name="Simple">
  <applicationmode name="OSDEFAULTAPPMODE">
    <autostarttasks><taskref name="UART"/></autostarttasks>
  </applicationmode>

  <task name="UART" function="functionTaskUART"
          priority="1" activation="1"/>

  <task name="CAN0" function="functionTaskCAN0"
          priority="2" activation="1"/>
...
  <osapplication name="theOneApp" trusted="true">
    <componentref name="LavA"/>
  </osapplication>

  <component name="LavA">
    <taskref name="UART"/>
    <taskref name="CAN0"/>
    ...
  </component>
</ciaoApp>
```

Quelltext 6.3: Auszug aus „config.xml“

Im Idealfall könnte eine Werkzeugkette das konfigurierte Betriebssystem samt Anwendung als Eingabe verarbeiten. Daraus würde dann ein anwendungsspezifisch maßgeschneidertes MPSoC generiert. Anschließend würde die Anwendung bzw. ihre *Threads* auf die einzelnen SoCs verteilt werden, sodass die Auslastung von Kommunikationsverbindungen und Rechenknoten im möglichst optimalen Bereich liegen.

Diese Arbeit, insbesondere die optimale Verteilung von *Threads*, ist im Rahmen einer Diplomarbeit jedoch nicht realisierbar. Daher muss nach der globalen Konfigurierung des einen Betriebssystems mit *pure::variants* die Anwendung manuell aufgeteilt werden. Im einfachsten Fall wird der gesamte Verzeichnisbaum für die Anzahl der gewünschten SoCs dupliziert. Danach muss in jedem Baum die kopierte Anwendung auf den jeweiligen *Thread* begrenzt werden. Optional ist eine nachträgliche Anpassung des Betriebssystems, etwa durch Entfernen von überflüssigen Gerätetreibern.

Nachdem dieser Prozess durchgeführt wurde, kann die entwickelte Werkzeugkette die Verzeichnisbäume verarbeiten und wie gewohnt die Hard- und Software erstellen bzw. ausführen. Die anfängliche Konfigurierung des Betriebssystems und die manuelle Bearbeitung der Anwendung veranlassen so eine durchgängige Maßschneiderung von Hard- und Software.

6.2 Simulation einer UART Applikation

Hier soll die Simulation eines MPSoCs mit vielen vernetzten SoCs betrachtet werden. Es wird untersucht, ob und inwiefern die Simulation bei der Entwicklung von MPSoCs und ihren Anwendungen hilfreich sein kann. Das folgende Kapitel 6.2.1 stellt die dazu verwendete Anwendung und ihre Simulation vor. Darauf folgt in Kapitel 6.2.2 eine Diskussion über den Simulator als alternative Plattform.

6.2.1 Die Anwendung

In Abbildung 6.1 wird die Beispielanwendung illustriert. Die Anwendung verteilt sich auf insgesamt 15 Rechenknoten, deren Prozessor die ZPU bildet. Um zusätzliche Komplexität einzubringen, wurde die Kommunikationsstruktur in Busse mit jeweils drei verbundenen Knoten aufgeteilt. Damit bildet das MPSoC einen vollständigen binären Baum der Tiefe vier.

Die Hardwarekonfiguration der Wurzelknoten und Blattknoten besteht aus einem IPC und einem UART. Die restlichen Knoten besitzen zwei IPCs, um jeweils mit der Ebene darüber sowie mit ihren beiden Kindknoten zu kommunizieren.

Das Ziel der Anwendung besteht darin, eine Eingabe über den UART des Wurzelknotens auf alle Blattknoten zu verteilen, welche das eingegebene Zeichen schließlich mit Hilfe ihres UARTs ausgeben. Diese Aufgabe ist – zugegebenermaßen – trivial und könnte durch einfachere Architekturen gelöst werden. Hier soll der Fokus jedoch einzig auf dem Umgang mit der Werkzeugkette liegen.

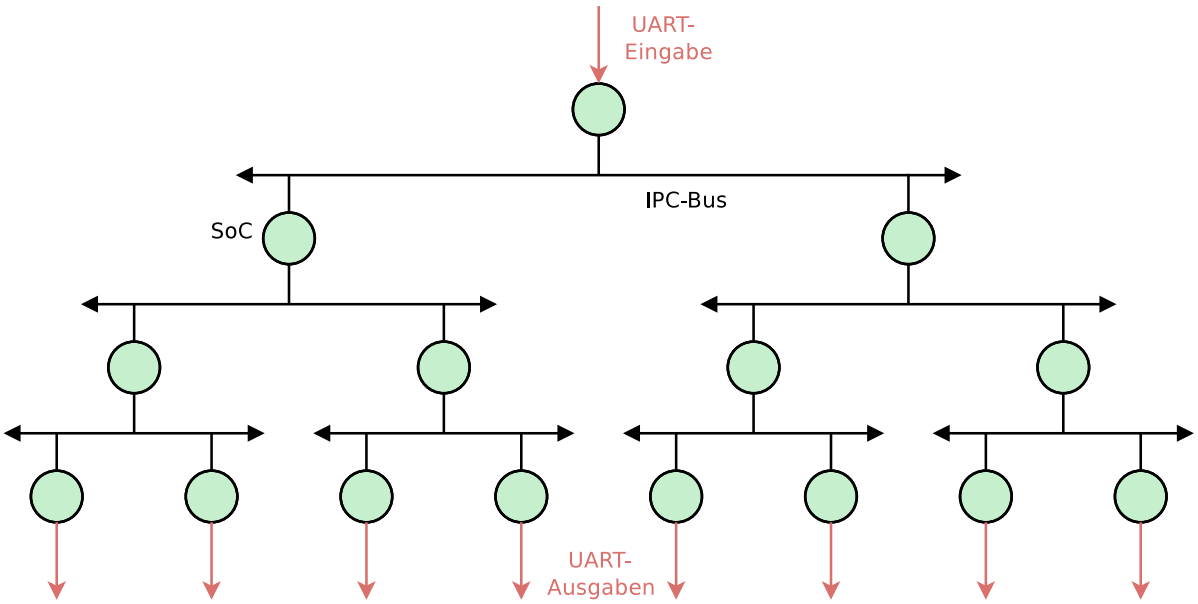

Abbildung 6.1: Beispielanwendung für die Simulation

Die Programmierung der Knoten unterscheidet sich grob in drei Gruppen:

1. Der Wurzelknoten: Der Wurzelknoten gibt die Aufforderung für die Eingabe eines Zeichens aus. Sobald ein Zeichen eingetroffen ist, wird dies an die zweite Ebene weitergereicht und startet damit die Kommunikation innerhalb des MPSoCs.

2. Knoten der mittleren Ebenen: Die Knoten der Ebenen zwei und drei warten, ähnlich dem Wurzelknoten, auf die Eingabe eines Zeichens. Das Zeichen wird über einen IPC empfangen und anschließend weitergereicht an die nächsttiefere Ebene.

3. Die Blattknoten: Die Blattknoten des Baums empfangen das Zeichen wiederum per IPC und geben es über ihren jeweiligen UART aus.

Diese drei Aufgaben können sehr leicht mit einzelnen *Threads* abgebildet werden. Wie bereits in Kapitel 6.1.3 angesprochen, müssen die *Threads* manuell aufgeteilt werden, um die Anwendung zu implementieren. Ebenso muss eine Maßschneiderung der Systemsoftware für jeden Knoten einzeln durchgeführt werden.
Die Werkzeugkette wird mit den Parametern *No Compile*, dem Verzeichnis, in dem die verteilte LavA-Anwendung zu finden ist und dem Simulator als gewünschte Zielarchitektur gestartet. Der Aufruf, mit allen Parametern, wird in Quelltext 6.4 gezeigt.

```
$ ./lava.sh -nc -src=$PWD/parse/eval_sim/ sim
```

Quelltext 6.4: Start der Werkzeugkette mit Zielarchitektur Simulator

Der Parser beginnt die Suche nach der Hardware API und erstellt letztlich das Modell des MPSoCs. Der Flächenbedarf der Schaltung wird hier nicht berechnet, da kein FPGA zum Einsatz kommt. Anschließend startet oAW mit der Generierung der Konfigurationsdatei für XVCL. Das Endergebnis des Arbeitsschritts ist zwar überflüssig, jedoch wird in dem Prozess eine Validierung des Modells durchgeführt, um eine, selbst für den Simulator, fehlerhafte Konfiguration auszuschließen.
Der letzte Schritt ist der Start des Simulators. Dieser interpretiert das generierte Modell und erstellt pro SoC ein Verzeichnis mit dessen SoC-Nummer, um dort z.B. die nötigen *FIFOs* anzulegen. Weiter werden die sieben benötigten Kommunikationsstrukturen durch ihre *Message Queues* simuliert, um eine Kommunikation zwischen den SoCs herzustellen. Quelltext 6.5 gibt die initiale Ausgabe des gestarteten Simulators wieder.

```
UART [SoC: 1, Nr: 0]   -  Enter character
```

Quelltext 6.5: Ausgabe zu Beginn der Simulation

Für weitere Aktivitäten innerhalb des MPSoCs muss eine Eingabe beim ersten SoC, dem Wurzelknoten, erfolgen. Dazu wird mit *echo* ein Zeichen in die Eingangs-*FIFO* des UARTs eingegeben, was Quelltext 6.6 darstellt.

```
$ echo -n A > ./1/uart0.rx
```

Quelltext 6.6: Eingabe eines Zeichens in den UART von SoC1

Danach fährt die verteilte Anwendung mit der Ausgabe in Quelltext 6.7 fort. Gut erkenntlich sind die unregelmäßigen Reihenfolgen bei den Ausgaben, die nun vom *Scheduler* des Hostsystems abhängen. So wird die Weiterleitung der Eingabe von Knoten fünf an dessen Kinder zehn und elf von der erneuten Aufforderung zur Eingabe eines Zeichens unterbrochen. Natürlich könnte die Anwendung so erweitert werden, dass erst alle Blattknoten die erfolgte Ausgabe zurück zum Wurzelknoten melden müssen, bevor dieser den nächsten Durchgang starten kann. Der Einfachheit halber wurde auf diese Koordination jedoch verzichtet.

```
IPC  [From: 1, To: 2]  -  65
IPC  [From: 2, To: 4]  -  65
IPC  [From: 2, To: 5]  -  65
IPC  [From: 5, To: 10] -  65
IPC  [From: 4, To: 8]  -  65
UART [SoC: 1, Nr: 0]   -  Enter character
IPC  [From: 5, To: 11] -  65
IPC  [From: 4, To: 9]  -  65
IPC  [From: 1, To: 3]  -  65
UART [SoC: 11, Nr: 0]  -  A
UART [SoC: 10, Nr: 0]  -  A
UART [SoC: 8, Nr: 0]   -  A
UART [SoC: 9, Nr: 0]   -  A
IPC  [From: 3, To: 6]  -  65
IPC  [From: 3, To: 7]  -  65
IPC  [From: 6, To: 12] -  65
IPC  [From: 6, To: 13] -  65
UART [SoC: 12, Nr: 0]  -  A
UART [SoC: 13, Nr: 0]  -  A
IPC  [From: 7, To: 14] -  65
IPC  [From: 7, To: 15] -  65
UART [SoC: 14, Nr: 0]  -  A
UART [SoC: 15, Nr: 0]  -  A
```

Quelltext 6.7: Ausgabe des Simulators nach der Zeicheneingabe

Abschließend zeigt ein *Screenshot* von *top* in Abbildung 6.2 die Ausführung aller SoCs durch eigenständige Prozesse unter Linux. Der Zeitpunkt des *Screenshots* ist direkt nach dem Start der Simulation. Der Wurzelknoten wartet in einer Endlosschleife auf die entsprechende Eingabe, weshalb der Prozess eine extrem hohe Prozessorlast erzeugt.

6.2.2 Bewertung der Simulation

Der Simulator bietet natürlich keine richtige Hardware, sondern simuliert diese lediglich. Zum Einsatz kommt dabei die Interprozesskommunikation des Betriebssystems Linux, etwa in Form von *FIFOs* oder *Message Queues*. Die Implementierung des Simulators ermöglicht es weiterhin, externe Programme anzuschließen. Diese könnten dann beispielsweise eine Verbindung zu einem realen CAN-Bus herstellen oder die Ein- und Ausgabe-*FIFOs* eines UARTs mit einer seriellen Schnittstelle verbinden.

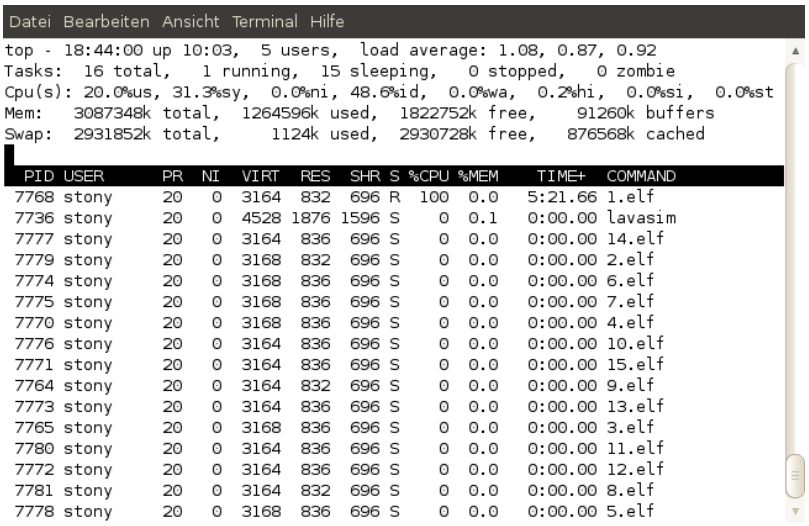

Abbildung 6.2: Ausgabe von *top* während der Simulation

Ein weiterer Punkt ist die begrenzte Parallelität der Simulation. Die eigentlich nebenläufig durchgeführten Aufgaben werden durch eigenständige Prozesse abgebildet. Dadurch ist die Ausführung zu einem gewissen Grad sequentialisiert worden und hängt von einigen Faktoren des Hostsystems ab. Die Anzahl der verfügbaren Rechenkerne stellt eine obere Schranke für die gleichzeitig simulierten SoCs dar. Zwischen mehreren Simulationen ist z.B. die Reihenfolge verschickter IPC-Nachrichten nicht eindeutig. Gründe hierfür könnten der *Scheduler*, die Startreihenfolge der Prozesse sowie Interprozesskommunikation sein. Diese Aspekte können zu variierenden Simulationen führen, wodurch Tests von Echtzeitanwendungen unmöglich werden.
Neben diesen Einschränkungen ist der Simulator dennoch in der Lage, für eine Plattform zu sorgen, auf der u.a. Kommunikationsprotokolle entwickelt und getestet werden können (siehe Kapitel 6.7 in [26]). Die Verarbeitung des MPSoCs durch Prozesse bietet dabei mehr Möglichkeiten, um Fehler zu finden. So können Zugriffe auf die simulierte Hardware über Statusmeldungen kenntlich gemacht werden. Weiter kann die Ausgabe in Dateien umgeleitet werden, um nachhaltige Protokolle zu liefern.
Der Verzicht auf die Hardware bietet hier die größten Vorteile. Potentielle Fehlerquellen werden drastisch reduziert. Weiter sind keine Resourcenbeschränkungen gegeben. Ein zusätzliches Gerät für Diagnosemeldungen kann kurzzeitig eingebaut und in der finalen Version wieder entfernt werden.
Gerade bei umfangreichen Systemen mit vielen Prozessoren und Geräten wird die Synthese sehr viel Zeit beanspruchen, in der keine weitere Arbeit möglich ist. In Tabelle 6.1 sind die Ausführungszeiten der einzelnen Arbeitsschritte aufgelistet. Ziel der Werkzeugkette war das Virtex-5. Die Berechnungen führte ein Intel Xeon E5345 durch. Diese CPU besitzt acht Rechenkerne mit jeweils 2,33GHz Taktgeschwindigkeit.
Die einzelnen Anwendungen nutzen das für LavA entwickelte Betriebssystem aus [26]. Dadurch enthält ein SoC-Verzeichnisbaum ca. 30 gewobene Quelltextdateien, die der Parser durcharbeiten muss. Auffällig ist die hohe Rechenzeit der Synthese, die in diesem Beispiel mehr als zehn mal so viel Zeit wie der Rest der Werkzeugkette beansprucht.

Aktion	Dauer (min:sec)[a]
Übersetzen einer SoC-Anwendung[b]	0:35
Parsen einer SoC-Anwendung	0:08
Parsen aller SoC-Anwendungen	2:13
SPC generieren	0:02
VHDL-Plattform generieren	0:04
Synthese für Virtex-5	23:45

Tabelle 6.1: Dauer der einzelnen Arbeitsschritte der Werkzeugkette

[a]Gemessen mit dem Unix Werkzeug *time*
[b]Enthält das Betriebssystem aus [26]

6.3 Kostenabschätzung von MPSoCs

In diesem Kapitel werden Ergebnisse der Kostenabschätzung von MPSoC-Modellen intensiver behandelt. Es wurden drei Testsysteme erstellt, deren Verbrauch an LUTs und Block RAMs zum einen mit der entwickelten und zum anderen mit Xilinx' Werkzeugkette berechnet wurden. Die MPSoCs eins und zwei entsprechen der Anwendung aus Kapitel 6.1. Dabei wurde ein Diagnosesystem mit einem CAN-Bus sowie ein *Gateway* mit vier CAN-Bussen erstellt. Das dritte MPSoC ist im vorigen Kapitel 6.2 beschrieben.

Die Hardware sieht im Detail wie folgt aus:

System	Prozessor	Peripherie
1	2 MBLite je 8KB RAM und Barrel Shifter	1 UART Baudrate 57600 1 CAN mit Filter 2 IPC Controller 32Bit
2	4 MBLite je 8KB RAM und Barrel Shifter	4 CAN mit Filter 4 IPC Controller 32Bit
3	15 ZPU je 2KB RAM	9 UART Baudrate 57600 21 IPC Controller 32Bit

Tabelle 6.2: Konfiguration der Testsysteme

Zuerst wird die Abschätzung der LUTs in Tabelle 6.3 betrachtet. Die Größenordnung der Abschätzung liegt bei den System 1 und 2 sehr nahe bei den tatsächlichen Werten. Eine maximale Abweichung von $+1,28\%$ ist noch vertretbar. Generell lässt sich mit einem nach oben abgeschätzten Wert besser arbeiten, da er eine unnötige Synthese verhindern kann.

Bei größeren Systemen wird ein zu geringer Verbrauch prognostiziert, da die Kostenfunktionen für die Komponenten SoC und MPSoC fehlen (siehe Kapitel 5.3.1), während die Komplexität der Verschaltung weiter ansteigt. Eine weitere Erkenntnis liegt darin, dass mit der Umstellung von LUTs mit sechs statt vier Eingängen keine Reduzierung des Flächenverbrauchs um ein Drittel stattfindet. Im Mittel wird der Bedarf an LUTs um 25% gesenkt, wenn ein Virtex-5 eingesetzt wird. Ein Grund hierfür könnte das *Routing* sein, welches die Einzelkomponenten des MPSoCs verschalten muss und hier gewisse konstante Kosten erzeugt.

System	Spartan-3			Virtex-5		
	geschätzt	gemessen	Differenz	geschätzt	gemessen	Differenz
1	7843	7793	+0,69%	5687	5615	+1,28%
2	17496	17577	-0,46%	12852	12897	-0,35%
3	23886	24755	-3,51%	17745	19508	-9,04%

Tabelle 6.3: Ressourcenverbrauch (LUTs) geschätzt und gemessen

In Tabelle 6.4 wird der Speicherverbrauch durch belegte Block RAMs aufgezeigt. Hier stimmen Messung und Abschätzung häufig überein. Lediglich für die MBLite-Systeme auf dem Virtex-5 werden niedrigere Werte gemessen. Xilinx' Werkzeugkette optimiert an dieser Stelle den Gebrauch der Block RAMs: Ein MBLite benötigt für die drei Registerbänke insgesamt drei Block RAMs, die jeweils nicht voll ausgelastet sind. Falls eine weitere MBLite-Instanz im MPSoC enthalten ist, wird die Synthese die insgesamt sechs Registerbänke in nur drei Block RAMs abbilden. Die Abschätzung hingegen betrachtet nur lokal die einzelnen Komponenten eines MPSoCs und errechnet so für jeden MBLite die volle Anzahl an benötigten Block RAMs.

System	Spartan-3			Virtex-5		
	geschätzt	gemessen	Differenz	geschätzt	gemessen	Differenz
1	15	15	0%	11	8	-27%
2	32	32	0%	24	18	-25%
3	15	15	0%	15	15	0%

Tabelle 6.4: Ressourcenverbrauch (Block RAM) geschätzt und gemessen

7 Zusammenfassung und Ausblick

In Kapitel 7.1 wird zunächst eine Beurteilung zu der durch LavA verfolgten Strategie abgegeben. Danach fasst Kapitel 7.2 die Leistungen und Erkenntnise dieser Diplomarbeit zusammenfassen. Abschließend werden in Kapitel 7.3 Ideen für die weitere Arbeit im Bereich der Maßschneiderung von MPSoCs durch Code-Analyse formuliert.

7.1 Beurteilung von LavA

Dieses Kapitel soll den Ansatz zur durchgängigen Konfigurierbarkeit von Hard- und Software durch das Projekt LavA diskutieren. Das allgemeine Ziel ist die Konfigurierbarkeit von Anwendung, Systemsoftware und Hardware mit den selben Mitteln, etwa *pure::variants*. In dieser Diplomarbeit wurde dazu die Maßschneiderung von MPSoCs durch Code-Analyse entwickelt, damit die Software die geforderte Hardware „deklarieren“ kann.
Passen Software und die abgeleitete Hardware optimal zueinander? Die realisierte Werkzeugkette untersucht die eingegebene Software und erstellt durch eine transparente Umsetzung ein MPSoC. Dieses MPSoC entspricht der geforderten Hardware, wie sie durch die Hardware API instanziiert wurde. Auf dieser Ebene kann obige Frage positiv beantwortet werden.
Darüber hinaus muss untersucht werden, ob die vorgefundene Software das Problem, für das sie eingesetzt wird, optimal löst. Die anfängliche Anwendung zur Lösung des Problems, im folgenden MPSoC-Anwendung genannt, wird als *Single-Core*-Anwendung programmiert. Sie basiert auf dem Betriebssystem CiAO und wird durch *Thread*s realisiert. Systemsoftware und MPSoC-Anwendung werden einmalig konfiguriert und anschließend in einzelne, kleinere Anwendungen separiert (vgl. Kapitel 6.1.3), im folgenden SoC-Anwendung genannt.
Natürlich kann die Konfiguration der Systemsoftware für die MPSoC-Anwendung nicht gleichzeitig optimal auf die einzelnen SoC-Anwendungen zugeschnitten sein. Beispielsweise wird das *Scheduling* überflüssig werden, wenn eine SoC-Anwendung lediglich aus einem *Thread* besteht. Ebenso können Gerätetreiber vorhanden sein, die nicht eingesetzt werden. Letzteres würde durch den Parser zu ungenutzter Hardware führen. Dadurch wäre das MPSoC zwar anwendungsgetrieben konfiguriert, jedoch nicht maßgeschneidert auf das eigentliche Problem.
Eine nachträgliche manuelle Konfiguration für jede SoC-Anwendung ist mühselig, vor allem, wenn ein möglichst optimales MPSoC per *Design Space Exploration* gesucht werden soll. Daher muss ein Automatismus eingefügt werden, der aus der globalen Konfiguration und der aktuellen SoC-Anwendung eine neue Konfiguration ableitet. Hier wird es

schwierig zu unterscheiden, welche Entscheidungen des Entwicklers während der Konfiguration der MPSoC-Anwendung revidiert werden dürfen. Letztlich führt kein Weg an diesem Prozess vorbei, um eine durchgängige Konfigurierbarkeit zu ermöglichen.

7.2 Zusammenfassung

In dieser Diplomarbeit wurde die Maßschneiderung von MPSoCs durch Code-Analyse entwickelt. Das Konzept der benötigten Werkzeugkette wurde in Kapitel 3 vorgestellt. Es wurden verschiedene Möglichkeiten diskutiert, wie der Bedarf von Hardware in der Softwarestruktur definiert werden kann. Daraus resultierte die Hardware API, eine auf C++ *Templates* basierende Abbildung der einzelnen Hardwarekomponenten. Diese wird modellgetrieben entwickelt, um Flexibilität für Erweiterungen und alternative Zielarchitekturen, wie FPGAs oder Simulatoren, zu bieten. Die entstandene Werkzeugkette liefert einen vollautomatisierten Prozess, welcher den Entwickler bei der Anwendungsentwicklung unterstützt. Die einzelnen Arbeitsschritte ermöglichen eine Analyse der verteilten Anwendung, Konfigurierung der MPSoC-Plattform und schließlich die Ausführung auf der Zielplattform.

In den Kapiteln 4 und 5 wurden die einzelnen Komponenten behandelt. Die zentrale Aufgabe kommt dem Parser zu, der die Instanzen der Hardware API und ihre Initialisierung aus dem Quelltext der Applikation finden und konfigurieren muss. Letztlich wird daraus die Struktur des anwendungsspezifischen MPSoCs erstellt. Die Eingabe des Parsers besteht aus der verteilten LavA-Anwendung, die in einzelne Verzeichnisse separiert wurde. Jedes Verzeichnis enthält den Quelltext für ein SoC. Die nächste Komponente ist die Ressourcenabschätzung der MPSoC-Struktur. Diese ermöglicht eine frühe Beurteilung, ob das spezifizierte FPGA genug Kapazität bietet, um die gewünschte Hardwareplattform aufzunehmen. Hierzu wurde ein Kostenmodell erstellt, welches den Flächenverbrauch der einzelnen Komponenten und ihrer Komposition ermittelt. Eingebettet ist die Ressourcenabschätzung in einer statischen Analyse des MPSoC-Modells, die weitere semantische Überprüfungen durchführt, um fehlerhafte Konfigurationen auszuschließen. Mit dem auf Linux basierenden Simulator wurde eine zweite Ausführungsplattform umgesetzt, die eine zeiteffiziente Entwicklung der Anwendung ermöglicht. Die einzelnen SoCs werden dabei durch eigenständige Prozesse simuliert und kommunizieren über Interprozesskommunikation.

Abschließend fand in Kapitel 6 eine Bewertung der Ergebnisse statt. Hier wurde die Benutzung der Werkzeugkette, die Ressourcenabschätzung und die Simulation exemplarisch vorgestellt und beurteilt.

Zusammen mit dem konfigurierbaren Betriebssystem aus [26] bildet diese Diplomarbeit ein Instrument für die Entwicklung von MPSoC-Applikationen und ihrer prototypischen Ausführung auf einem FPGA oder Simulator.

7.3 Ausblick

Bisher bildet die implementierte Werkzeugkette eine transparente Eins-zu-eins-Umsetzung der geforderten Hardware. Zukünftige Arbeiten sollten hier ansetzen und die Struktur des MPSoCs auf funktionale und nicht-funktionale Eigenschaften hin untersuchen und entsprechend optimieren. Als Vorbild kann die Arbeit aus [6] dienen.
Einen zweiten Aspekt bilden die Anwendungsfälle des LavA-Projekts. Damit die entwickelten Werkzeuge besser beurteilt werden können, müssen aussagekräftige Anwendungen gefunden und realisiert werden. Die Einsatzgebiete können sich über die Gebiete Multimedia wie auch Kontrollsysteme erstrecken. Ein Beispiel für Letztere könnte die Neuimplementierung der Arbeit der Projektgruppe CoaCh [34] sein, die bereits MPSoCs im automotiven Bereich einsetzte.

Literaturverzeichnis

[1] Spinczyk, Olaf ; Engel, Michael ; Meier, Matthias: *Laufzeitplattform für anwendungsspezifische verteilte Architekturen*. http://ess.cs.tu-dortmund.de/DE/Research/Projects/LavA/index.html, 2009–2010

[2] Spinczyk, Olaf ; Lohmann, Daniel ; Urban, Matthias: AspectC++: An AOP Extension for C++. In: *Software Developers Journal* (2005), Mai, Nr. 5, S. 68–76

[3] 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 USENIX Annual Technical Conference*. Berkeley, CA, USA : USENIX Association, Juni 2009, S. 215–228

[4] Hofer, Wanja: *Aspect-Oriented Design and Implementation of an AUTOSAR-Like Operating System Kernel*, Universität Erlangen-Nürnberg, Diplomarbeit, 2007

[5] Meier, Matthias ; Engel, Michael ; Steinkamp, Matthias ; Spinczyk, Olaf: LavA: An Open Platform for Rapid Prototyping of MPSoCs. In: *Proceedings of the 20th International Conference on Field Programmable Logic and Applications (FPL '10)*. Milano, Italy : IEEE Computer Society Press, 2010. – to appear

[6] Thompson, Mark ; Nikolov, Hristo ; Stefanov, Todor ; Pimentel, Andy D. ; Erbas, Cagkan ; Polstra, Simon ; Deprettere, Ed F.: A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs. In: *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. New York, NY, USA : ACM, 2007. – ISBN 978–1–59593–824–4, S. 9–14

[7] Kangas, Tero ; Kukkala, Petri ; Orsila, Heikki ; Salminen, Erno ; Hännikäinen, Marko ; Hämäläinen, Timo D. ; Riihimäki, Jouni ; Kuusilinna, Kimmo: UML-based multiprocessor SoC design framework. In: *ACM Trans. Embed. Comput. Syst.* 5 (2006), Nr. 2, S. 281–320. – ISSN 1539–9087

[8] Stahl, Thomas ; Völter, Markus ; Efftinge, Sven ; Haase, Arno: *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. 2. Heidelberg : dpunkt, 2007. – ISBN 978–3–89864–448–8

[9] Object Management Group: *UML: Unified Modeling Language™*. http://www.uml.org, 2005

[10] *openArchitectureWare*. http://www.openarchitectureware.org, 2005–2008

[11] Eclipse Foundation: *Eclipse*. http://www.eclipse.org, 2000–2010

[12] Urban, Matthias ; Lohmann, Daniel ; Spinczyk, Olaf: The aspect-oriented design of the PUMA C/C++ parser framework. In: *AOSD '10: Proceedings of the Eighth International Conference on Aspect-Oriented Software Development*. New York, NY, USA : ACM, 2010. – ISBN 978–1–60558–958–9, S. 217–221. – Industry Track

[13] Free Software Foundation: *GNU General Public License*. http://www.gnu.org/licenses, 2008

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

[15] Meier, Matthias: *Merkmalbasierte statische Konfigurierung von MPSoCs*, Technische Universität Dortmund, Diplomarbeit, Mai 2009. http://ess.cs.tu-dortmund.de/Teaching/Theses/2009/DA_Meier_2009.pdf

[16] Rhoads, Steve: *Plasma MIPS*. http://opencores.org/project,plasma, 2001–2010

[17] Kranenburg, Tamar: *MB-Lite*. http://opencores.org/project,mblite, 2009–2010

[18] Harboe Øyvind: *ZPU*. http://opencores.org/project,zpu, 2008–2009

[19] *GCC, the GNU Compiler Collection*. http://gcc.gnu.org, 1987–2010

[20] Kahn, G.: The semantics of a simple language for parallel programmingc. In: Rosenfeld, J. L. (Hrsg.): *Information processing*. Stockholm, Sweden : North Holland, Amsterdam, Aug 1974, S. 471–475

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

[22] *eCos*. http://ecos.sourceware.org, 2010

[23] Alexandrescu, Andrei: *Modern C++ design: generic programming and design patterns applied*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2001. – ISBN 0–201–70431–5

[24] Vandevoorde, David ; Josuttis, Nicolai M.: *C++ Templates: The Complete Guide*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2002. – ISBN 0–201–73484–2

[25] Xilinx: *Webauftritt*. http://www.xilinx.com, 1984–2010

[26] Vogt, Stephan: *LavA OS: Ein Betriebssystem für konfigurierbare MPSoCs*, Technische Universität Dortmund, Diplomarbeit, 2010

[27] xmlsoft.org: *The XML C parser and toolkit of Gnome, libxml2*. http://www.xmlsoft.org,

[28] Object Management Group: *XML Metadata Interchange*. http://www.omg.org/spec/XMI/index.htm, 2007

[29] Gingol, Tristan: *GHDL*. http://ghdl.free.fr, 2008

[30] Bybell, Tony: *GTKWave*. http://gtkwave.sourceforge.net, 2006–2010

[31] Xilinx: *Spartan-3E FPGA Family: Data Sheet*. 2009

[32] Foundation, Free S.: *GNU bourne-again shell, Bash*. http://tiswww.case.edu/php/chet/bash/bashtop.html, 1989–2010

[33] Pure-Systems GmbH: *pure::variants User's Guide, Version 3.0*. http://www.pure-systems.com/fileadmin/downloads/pure-variants/doc/pv-user-manual.pdf, 2008

[34] Austin, David ; Ben-Shlomo, Adrian ; Borchert, Christoph ; Bostan, Elena-Crina ; Boris Golubovic, Jens K. ; Mertens, Christoph ; Niewerth, Jan-Philipp ; Pyka, Arthur ; Schindler, Christian ; Steinkamp, Matthias ; Zou, Jiong: Projektgruppe 533 Car on a Chip, CoaCh / Technische Universität Dortmund. 2009. – Abschlussbericht

Abbildungsverzeichnis

1.1 Schema des LavA-Projekts (Quelle: [1]) 2

2.1 Das *Ecore*-Metametamodell (Quelle: [11]) 6
2.2 Übersicht über den Designprozess mit Daedalus (Quelle: [6]) 9
2.3 Übersicht über den Designprozess mit Koski (Quelle: [7]) 11

3.1 Das Metamodell der Hardware API 19
3.2 Die Struktur der SoCs 20
3.3 Unterschiedliche Implementierungen der Hardware API 25
3.4 Skizze der Werkzeugkette 26
3.5 Von der Eingabe zum MPSoC-Modell 27
3.6 Werkzeugkette für die Implementierung auf Hardware 28

4.1 Modellierung der Datenhaltung im Parser 37
4.2 Aufbau eines Xilinx FPGAs (Quelle: [31]) 45

5.1 Verteilung der Programmzeilen auf die Kernbereiche des Parsers 48
5.2 Vergleich der Kommunikation auf Hardware und per Simulation 51
5.3 LUT-Verbrauch des Interrupt Controllers 55

6.1 Beispielanwendung für die Simulation 64
6.2 Ausgabe von *top* während der Simulation 67