8. E.I.S.-Workshop am 8. und 9. April 1997 an der Universität Hamburg

Objekt-orientierte Modellierung von hybriden Hardware-Software-Systemen am Beispiel des "European Home System" (EHS) Standards

Andreas Koch
Abt. Entwurf integrierter Schaltungen (E.I.S.), TU Braunschweig
koch@eis.cs.tu-bs.de

Wir untersuchen die Anwendbarkeit von objekt-orientierten (OO-) Entwurfstechniken auf den HW- und HW-nahen SW-Entwurf. Dazu werden nach einer allgemeinen Einführung in OO-Konzepte Möglichkeiten zur Anwendung im VLSI-Entwurf, insbesondere auch in Form von erweiterten HDLs, vorgestellt. Als Beispiel für einen HW-nahen SW-Entwurf werden anschließend die bei einer OO-Modellierung des EHS-Standards gesammelten Erfahrungen zusammengefaßt.

1 Einleitung

Mit der stetigen Fortentwicklung von Fertigungstechnologien und EDA-Werkzeugen können immer größere Schaltungen technisch zuverlässig hergestellt werden. Dabei erlaubt die heute erreichte Integrationsdichte die Implementierung von ganzen Systemen auf einem Chip.

Leider haben sich die Entwurfsmethoden selbst nicht in gleichem Maße wie Fertigung und Werkzeuge verbessert. Selbst bei Einsatz von Hardware-Beschreibungssprachen wie Verilog und VHDL gestaltet sich der Umgang mit komplexen Systemen, wie sie beispielsweise in Multimedia-Anwendungen auftreten, als schwierig, insbesondere beim Entwurf von Hardware (HW) mit Software (SW).

In dieser Hinsicht vollzieht der Hardware-Bereich eine in der Software-Produktion schon länger anhaltende Entwicklung nach. Dort ist schon lange nicht mehr die HW-Leistungsfähigkeit der begrenzende Faktor, sondern vielmehr die vergleichsweise grenzenlos gewachsene SW-Komplexität. Obwohl im SW-Bereich auch nach jahrzehntelanger intensiver Forschung immer noch kein Allheilmittel (silver bullet) zur Beherrschung der Komplexität gefunden wurde (und wohl auch nie gefunden werden wird), so haben sich doch Leitlinien und Empfehlungen kristallisiert, die den SW-Designer bei seiner Tätigkeit unterstützen können.

In Anbetracht der verwandten Problematik liegt es nahe, die Anwendbarkeit von Methoden des Software-Engineering (SE) auch zur Beschreibung von HW- bzw. hybriden HW- / SW- Systemen zu untersuchen [Quib95].

In dieser Arbeit werden einige Ansätze und Überlegungen vorgestellt, mit denen objekt-orientierte Methoden auch im VLSI-Bereich und der hardwarenahen Programmierung eingesetzt wurden. Zum letztgenannten Bereich wurde ein praktisches Experiment im Rahmen eines Projekts an der Abteilung E.I.S. durchgeführt. Dabei wurden Kernaspekte der Steuersoftware des European Home Standard (EHS) von zwei Arbeitsgruppen einmal klassisch prozedural und einmal objekt-orientiert modelliert.

2 Klassische Sichtweise

Der klassische Aufbau von Modellen sowohl im HW als auch im SW-Bereich basiert im wesentlichen auf den Punkten [Booc94]:

Abstraktion Reduziert ein Konzept auf seine essentiellen Charakteristika. Abstraktionen sind von der Perspektive des Betrachters abhängig (z.B. SPICE-Parameter, Pad-Positionen, etc. derselben Zelle).

Kapselung Versteckt die zugrundeliegende Struktur einer Abstraktion vor externen Betrachtern und erlaubt so ihren Austausch (z.B. Zelle mit gleichen elektrischen Daten, aber auf anderem Prozeß).

Modularität Die Aufteilung eines komplexen Modells in Sammlungen von zusammengehörigen Abstraktionen (z.B. die Partitionierung eines Prozessors in Datenpfad und Steuereinheit).

Hierarchie, Aggregation Die stufenweise Zerlegung eines Konzepts in Unterkonzepte, die selbst wiederum Unterkonzepte enthalten können (z.B. Aufbau eines Registerfeldes aus einzelnen Flip-Flops).

Typisierung Klassifiziert Konzepte und macht so Vorschriften über die gegenseitige Austauschbarkeit. Die Typisierung ist abhängig von der gewählten Abstraktion (z.B. sind ein BiCMOS und ein NMOS Inverter auf Netzlistenebene vom selben Typ (gegeneinander austauschbar), auf Layout und SPICE-Ebene haben sie aber unterschiedliche Typen).

Nebenläufigkeit Mehrere parallele Ausführungsfolgen, die sich nicht oder nur kontrolliert beeinflussen sollten (z.B. mehrere Gatter, die alle unabhängig, aber gleichzeitig schalten).

Diese Punkte sind Bestandteil aller gängigen Entwurfsmethoden, und haben, wie durch die Beispiele oben angedeutet, auch im HW/VLSI Bereich schon lange erfolgreich Einzug gehalten.

3 Objekt-orientierte Sichtweise

Was hat es nun mit der objekt-orientierten Sichtweise auf sich? Sie umfaßt alle der oben aufgezählten Aspekte der klassischen Sichtweise, *erweitert* sie aber noch um[1]

[1] Es gäbe noch weitere Aspekte (strong/weak typing, late/early binding, polymorphism), die für den reinen HW-Bereich aber nur von eingeschränkter Bedeutung sind.

Hierarchie, Vererbung Erlaubt die Herstellung von Verwandtschaftsbeziehungen zwischen Abstraktionen durch Angabe der Unterschiede (z.B. ist ein rücksetzbarer Zähler ein Zähler, der sich bei Anliegen des RESET-Signals auf den Wert 0 setzt).

Man beachte die unterschiedlichen Verwendungen von Hierarchie: Die Aggregation modelliert "ist-Teil-von" Beziehungen, die Vererbung modelliert "ist-eine-Art-von" Beziehungen. Zur Terminologie: Eine *Klasse* beschreibt die Struktur und das Verhalten einer Menge von Objekten, ihre *Instanzen*. Sie ist einem abstrakten Datentyp (ADT) vergleichbar. Darüberhinausgehend kann eine Klasse aber durch Vererbung aus einer allgemeineren Klasse durch Angabe der Unterschiede abgeleitet werden. Eine solche abgeleitete Klasse, auch *Unterklasse* genannt, spezifiziert diese Unterschiede i.d.R. durch die Definition weiterer Datenfelder und/oder Operationen (sog. *Methoden*), oder durch die Modifikation bestehender Methoden. Der Vorgang der Ableitung einer verfeinerten aus einer allgemeinen Klasse heißt *Spezialisierung*, das Zusammenfassen der Gemeinsamkeiten mehrerer Klassen zu einer gemeinsamen *Oberklasse* heißt *Generalisierung*.

Der objekt-orientierte (OO-) Ansatz hat seinen Ursprung in der SW-Entwicklung. Eine der ersten Programmiersprachen, die die OO- Formulierung von Algorithmen erlaubte, war Simula-67. Ihre Autoren haben das mächtige Konzept der Vererbung quasi zufällig beim Umgang mit komplexen Simulationsproblemen entdeckt. Damals wurde aber noch nicht von "objekt-orientierter Programmierung" oder gar "objekt-orientiertem Entwurf" gesprochen. Erst einige Jahre später führte die gewachsene Erfahrung beim Umgang mit dem neuen Konzept zur Entwicklung konsequent objekt-orientierter Sprachen (z.B. Smalltalk-72). Neue Entwicklungsmethoden, die abweichend von den damals vorherrschenden strukturierten Verfahren eine objekt-orientierte Sichtweise anwenden, kamen erst noch später auf. Erste Ansätze lassen sich bis ins Jahr 1983 zurückverfolgen [JCJO92].

Um den in der SW-Gemeinschaft mittlerweile erarbeiteten umfangreichen Erfahrungsschatz beim Umgang mit OO-Techniken besser nutzen zu können und die Wiederverwendbarkeit von Konzepten auch auf immer höheren Abstraktionsebenen zu ermöglichen, wurde in jüngster Zeit sehr erfolgreich das Mittel des *Entwurfsmusters* (design pattern) entwickelt [GHJV95]. Dabei wurden eine größere Menge von OO-Entwürfen auf wiederkehrende Muster untersucht, diese aus dem problemspezifischen Kontext extrahiert und verallgemeinert. Abschließend wird für jedes der so erkannten Muster noch eine Dokumentation, die auch Beziehungen zu verwandten Mustern aufzeigt, in einem einheitlichen Stil verfaßt. Dem SW-Entwickler steht so ein Katalog von nach Anwendungsbereichen kategorisierten Abstraktionen zur Verfügung, die sich leicht wiederverwenden lassen.

Um überzogenen Erwartungen vorzubeugen muß spätestens jetzt klargestellt werden: Der objekt-orientierte Ansatz ist nicht per-se mächtiger als der klassische. Jedes Problem, das sich objekt-orientiert beschreiben und lösen läßt, läßt sich auch klassisch lösen. Aber ein OO-Ansatz kann wesentlich eleganter (kürzer, leichter pflegbar, besser wiederverwendbar, ...) als ein klassischer Ansatz sein. Unglücklicherweise kann auch das Gegenteil eintreten: Durch die gesteigerte Ausdrucksfähigkeit eines OO-Modells können bei ungeschicktem Vorgehen selbst einfache Konzepte durch Konvolute beschrieben werden, deren Überschau- und Wiederverwendbarkeit noch weit unter der eines klassischen Modells liegen. Die objekt-orientierte Sichtweise ist lediglich ein Werkzeug, es bleibt dem Anwender überlassen, wie geschickt er damit umgeht.

Abbildung 1 ist ein Beispiel für ein OO-Modell in Booch-Notation [Booc94]. Die Klasse `Shape` hat ein Attribut `areacost` und die Methoden `area` und `price`, wobei der Preis das Produkt von `areacost` und `area` ist. Die Berechnung von `area` ist auf dieser Abstraktionsebene noch unbestimmt. `Shape` wird durch die Unterklassen (dargestellt durch Pfeile) `Circle`, `Rectangle` und `Triangle` spezialisiert. Jede der Unterklassen definiert weitere Attribute für ihre Ausmaße und implementiert eine entsprechende Methode zur Flächenberechnung. Bei Ausführung von `r.price()`, wobei `r` eine Instanz von `Rectangle` sein soll, würde also die Fläche des Rechtecks berechnet und zur Preiskalkulation verwendet. `Circle` implementiert eine weitere Verfeinerung: Wir nehmen an, daß der Preis für einen Kreis mit 10% Aufschlag versehen werden soll (Verschnitt). Dazu wird in `Circle` auch noch die `price()` Methode spezialisiert: Durch `super::price` wird zunächst die Preismethode der Oberklasse `Shape` ausgeführt, und das so erhaltene Ergebnis mit dem Aufschlag versehen. Schon bei diesem einfachen Beispiel wird die Wiederverwendbarkeit von Konzepten durch Angabe der Unterschiede deutlich: Alle Formen erben z.B. das Attribut `areacost` von `Shape`. Und auch die `price`-Methode wird nur dort verändert implementiert (`Circle`), wo sie sich von der geerbten Variante unterscheidet. Die Aggregationsbeziehung "eine `Shape` hat eine `Color`" wird durch die `color`-Beziehung modelliert.

4 Objekt-orientierte Methoden im HW/VLSI-Bereich

Nach dem großen Erfolg von OO-Ansätzen in der reinen SW-Entwicklung wurde auch bald über Anwendungsmöglichkeiten in bezug auf HW-nahe Applikationen nachgedacht. Interessanterweise ist dabei folgender Umstand eingetreten: In den Anfangszeiten des OO-SW-Entwurfs wurde der Wunsch nach sogenannten "Software-ICs" laut [Cox86]. Diese Software-ICs sollten eine Wiederverwendbarkeit ähnlich der von Standard-ICs (74xx etc.) haben. Nach jahrelangen Verzögerungen ist dieser Wunsch in einigen Bereichen schon fast erfüllt worden: Durch die Einführung von Entwurfsmustern und einem stetig wachsenden Angebot verschiedener frei verfügbarer oder kommerzieller Klassenbibliotheken aus den unterschiedlichsten Anwendungsbereichen (Grafik, Datenbankzugriffe, Kommunikation, Mathematik, ...) steht einem SW-Entwickler ein umfangreicher Katalog an wiederverwendbaren Abstraktionen zur Verfügung, die er aber durch Vererbung leicht an seine speziellen Anforderungen anpassen kann.

Im HW-Bereich, gerade in der VLSI-Entwicklung, ist diese Flexibilität aber leider noch nicht erreicht. Zwar existieren kommerzielle Zell-Bibliotheken, Generatoren und lizensierbare Funktionseinheiten (cores) aus diversen Anwendungsgebieten (DSP, CPU, RAM, Arithmetik, ...). Die Anpassungsmöglichkeiten an spezielle Anforderungen sind aber bei all diesen Alternativen (sofern sie nicht in Quellform, z.B. als HDL-Beschreibung oder Netzliste vorliegen), stark begrenzt und beschränken sich i.d.R. auf die Angabe

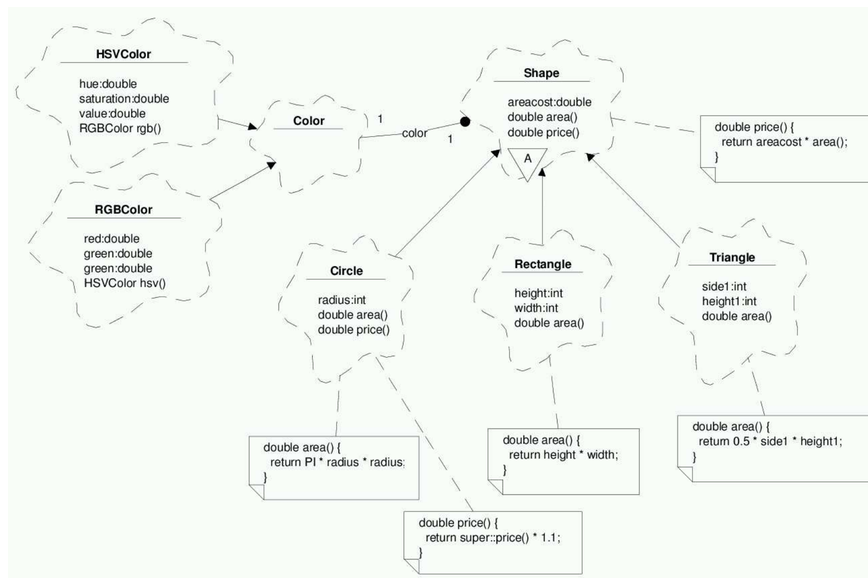

Abbildung 1: Klassendiagramm in Booch-Notation

von Parametern wie Bus-Breiten, Wait-States etc. Die im SW-Bereich vorhandene Möglichkeit, gezielt durch Vererbung Teilverhalten zu ändern, ohne den Zugriff auf die Quellen der Oberklasse zu haben, fehlt hier völlig. So hat sich der an den damaligen Stand der HW-Entwicklung angelehnte Wunsch der SW-Gemeinschaft schon fast erfüllt, während die HW-Gemeinschaft selbst heute mit ähnlichen Problemen konfrontiert ist.

Beim Einsatz von OO-Methoden im Bereich VLSI-Design sind grundsätzlich zwei unterschiedliche Ansätze zu betrachten.

4.1 Beschreibung von Strukturen

Hier werden die OO-Techniken zum Aufbau einer strukturellen Beschreibung von HW-Eigenschaften genutzt, nicht aber zur direkten Spezifikation von Verhalten. Beispiele hierfür sind C.S.L. [HWSW91] und OO-ADS [WWNS91], beides Sprachen zur Layout-Beschreibung. Ein in ihnen abgefaßtes OO-Programm generiert nach Eingabe von Werten für Parameter ein entsprechendes Layout. In Abbildung 2 wird gezeigt, wie durch Vererbung eine Spezialisierungshierarchie von einem allgemeinen Basisgatter (z.B. einem Buffer) über die Schaltungstechnologie (BiCMOS, CMOS, ...) bis zu einem konkreten Prozeß (ES2 CMOS, Toshiba CMOS, ...) aufgebaut wird. Zu den verfeinerten Methoden gehört beispielsweise das automatische Anpassen (sizing) von Transistorverhältnissen an Ausgangslasten, das abhängig von Technologie und Prozeß unterschiedlich, aber unabhängig von der Topologie des Layouts, vorgenommen wird. So muß bei Einführung eines neuen Prozesses lediglich eine neue Klasse mit entsprechender `size()` -Methode neu definiert werden. Die die Topologie bestimmenden Methoden der Oberklassen bleiben unverändert und vererben sich in die neue Unterklasse.

Der Einsatz von OO-Techniken bei der Beschreibung von Schaltungsstrukturen ist nicht von der Einführung neu-

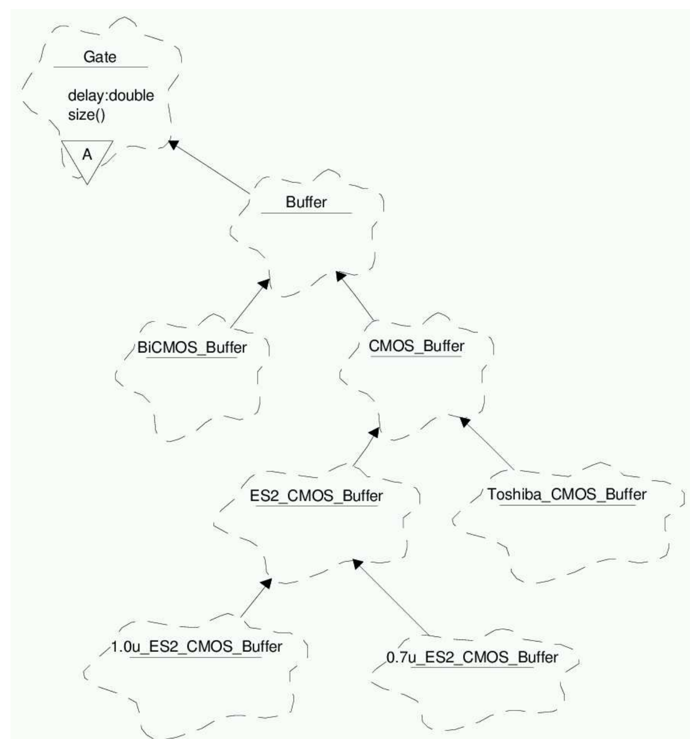

Abbildung 2: Nutzung von Vererbung in C.S.L. [HWSW91]

er, spezialisierter Sprachen (wie C.S.L. und OO-ADS) abhängig. So wurde beispielsweise in [DiMo92] [MoDi95] ein Vorgehen zum Entwurf eines 300MHz/115W ECL RISC-Prozessors beschrieben, bei dem das komplette Prozessorlayout durch ein C++ - Programm beschrieben wurde. Mittels geeigneter Vererbung wurde die Exploration des Lösungsraumes durch die Entwickler vereinfacht, indem lediglich Änderungen relativ zu bereits evaluierten Designalternativen spezifiziert wurden.

Dieser Einsatz von OO-Technologie im VLSI-Bereich unterscheidet sich kaum von allgemeinen SW-Anwendungen: Die OO-Methodik ist lediglich ein Mittel zum Zweck der Erstellung einer statischen Chip-Beschreibung (Netzliste oder Layout). Zwar können auch hier die Vorzüge des OO-Ansatzes ausgenutzt werden, die OO-Komponente kann aber weder direkt simuliert werden, noch kann mit ihr direkter Einfluß auf das Verhalten des Chips genommen werden. Beides geschieht immer über den Umweg der generierten statischen Chip-Beschreibung.

4.2 Beschreibung von Verhalten

Beim Versuch, OO-Techniken direkt bei der Verhaltensbeschreibung von HW anzuwenden, liegt es am nächsten, bestehende HDLs um entsprechende Konstrukte zu erweitern. Bei der Auswahl dieser Konstrukte ist eine gewisse Sorgfalt geboten, da sich das dynamische Anlegen und Löschen von Objekten, das in reinen SW-Lösungen und Entwurfsmustern routinemäßig stattfindet, nicht in HW abbilden läßt (es würde zum Entstehen und Verschwinden von Funktionseinheiten auf dem laufenden Chip führen). Ebenso kann auf einem Chip keine dynamische Typisierung (late binding, polymorphism) einer Funktionseinheit stattfinden, ihr Typ und Verhalten ist durch die Fertigung festgelegt. Durch die Beschränkung auf einfache Vererbung (wie in Abschnitt 3 angedeutet) verliert man zwar viel von der aus dem SW-Bereich bekannten Ausdrucksfähigkeit, gewinnt aber immer noch gegenüber klassischen HDLs.

Die Erweiterung von HDLs gestaltet sich als etwas schwieriger als die Erweiterung konventioneller Programmiersprachen (z.B. C zu C++), da das Verhalten von Hardware-Objekten (VHDL `entity`, Verilog `module`) nicht nur durch eine Menge von Funktionen bestimmt wird, sondern i.d.R. durch nebenläufige Prozesse realisiert ist (`process` in VHDL, `always` in Verilog). Bisherige Ansätze beschränken sich daher häufig auf die Vererbung von strukturellen Eigenschaften wie Ports, Attributen (z.B. Bus-Breiten) oder die Kardinalitäten von Aggregationsbeziehungen [AgMS95]. Eine direkte Modifikation von geerbtem Verhalten ist bei diesem Ansatz nicht möglich.

Nach jahrelangen Vorarbeiten (IEEE Working Group seit 1993) sind mittlerweile erste Implementierungen verfügbar, die, u.a. durch Eingriffe auf Prozeßebene, eine weitergehende Beeinflussung zulassen. Bei dem in [Vist96] vorgeschlagenen OO-VHDL können in der `architecture` einer Unterklasse neue Prozesse hinzugefügt oder durch gleiche Namensgebung benannte Prozesse (in VHDL möglich) der Oberklasse in Unterklassen ersetzt werden. Ein Beispiel für dieses Vorgehen ist die Erweiterung eines einfachen NAND-Gatters um eine transitionszählende Funktion in Abbildung 3.

Wie man sieht, wurde die gesamte Funktionalität des ursprunglichen NAND-Gatters `nand_gate` in der Unterklasse `nand_gate_acct` wiederverwendet, ohne daß dazu Programmtext kopiert werden mußte (wartungserschwerend). Prozesse sind also in HDLs ein sinnvoller Ansatzpunkt für Eingriffe in das Verhalten von Hardware-Objekten. Die Art des Eingriffs ist in OO-VHDL allerdings eingeschränkt: Ein Prozeß kann entweder im Ganzen aus einer Oberklasse übernommen werden (in Abbildung 3 `pnand`), im Ganzen überschrieben werden (nicht gezeigt), oder komplett neu hinzukommen (in Abbildung 3 `paccount`).

Die Möglichkeit, einen Prozeß aus der Oberklasse auch in der Unterklasse, ggf. in Modifikationen eingebettet, ablaufen zu lassen (ähnlich der `price`-Methode von `Circle` in Abbildung 1), fehlt in OO-VHDL. Dieses Problem ließe sich durch die folgende Erweiterung beheben: Durch Einführung der Syntax "`super` *Prozeßname_aus_Oberklasse*" (in Anlehnung an Smalltalk-80) sollte der dem so benannten Prozeß zugeordnete Anweisungsblock (body) an der Stelle der `super`-Anweisung ausgeführt werden. Dazu müssen auch die sensitivity lists der die `super`-Anweisung umschließenden Prozesse um die in der Oberklasse beobachteten Signale erweitert werden. Ein Beispiel für eine solche Anwendung wird in Abbildung 4 skizziert.

`count8` ist ein einfacher 8-bit Zähler (zur Vereinfachung haben wir auf `generic` verzichtet), dessen Zählverhalten durch den Prozeß `pcount` implementiert ist. Um nun diesen Zähler durch eine asynchrones Reset-Verhalten zu erweitern, wird in der Unterklasse `count8_r` der `pcount`-Block umdefiniert: Die Reset-Behandlung wird hinzugefügt, aber durch die `super pcount`-Anweisung das ursprüngliche Zählverhalten beibehalten. In OO-VHDL wäre dies nicht möglich gewesen: Wenn der gesamte `pcount`-Block ersetzt worden wäre, hätte auch in `count8_r` das Zählverhalten neu implementiert werden müssen. Falls einfach ein weiterer Prozeß zur Realisierung des Reset-Verhaltens hinzugefügt worden wäre, wäre durch die undefinierte Abarbeitungsreihenfolge entweder der Reset- oder der ZählProzeß zuerst ausgeführt worden. Im Falle eines asynchronen Resets wäre dieses Verhalten zwar unkritisch (Zeitunterschied von einem Delta bei der Simulation), in anderen Fällen könnte es aber zu Inkonsistenzen kommen. An diesem Beispiel wird auch deutlich, daß die sensitivity list eines `super` verwendenden Prozesses um die sensitivity-list des Prozesses in der Oberklasse ergänzt werden muß. Das Zählverhalten ist von `CLOCK` abhängig, während das Reset-Verhalten von `CLEAR` gesteuert wird. Um also einen korrekten Ablauf des Zählverhaltens in `count8_r` zu ermöglichen, wird die sensitivity-list von `pcount` hier implizit auf `CLEAR, CLOCK` erweitert. Ein analoges Vorgehen wird verwendet, um in `count8_e` das Zählverhalten durch ein `ENABLE`-Signal zu steuern.

Diese OO-VHDL-Erweiterung kann durch einen geeigneten Präprozessor wieder in Standard-VHDL übersetzt werden (Beispiel in Abbildung 5). Die Übersetzung geschieht statisch, der so entstandene VHDL-Code weist keinen Unterschied mehr zu klassisch prozeduralem Code auf. Auf diese Weise ist der gelegentlich erhobene Vorwurf, OO-entworfene HW sei automatisch ineffizient, von vornherein entkräftet. Wie bereits in Abschnitt 3 erwähnt: Das objektorientierte Vorgehen (welches hier nur zur Organisation des Quellcodes und Erhöhung der Wiederverwendbarkeit benutzt wird) ist nur ein Werkzeug, das von sich aus keinen Einfluß auf Effizienz des endgültigen Produktes haben sollte.

```
-- ************************************************************
-- basic nand gate
-- ************************************************************
entity nand_gate is
        port (in1, in2 : in Bit; output : out Bit);
end nand_gate;

architecture nand_behavior of nand_gate is
begin
        pnand: process(in1, in2) -- implement nand behavior
        begin
                out <= not (in1 and in2);
        end process;
end nand_behavior;

-- ************************************************************
-- a nand gate extended with transition accounting
-- ************************************************************

entity nand_gate_acct is new nand_gate
  -- defines a subclass of nand_gate
  -- all ports are inherited from nand_gate. we could have
  -- defined new ones specific to this entity, but it
  -- was not necessary in this example.
end nand_gate_acct;

architecture acct_behavior of nand_gate_acct is new nand_behavior
begin
  -- The process block L1, defined in the
  -- nand_behavior body, is inherited.

  -- Count the number of times we're executed.
  paccount: process -- implement accounting behavior
          variable count: integer := 0
  begin
    wait on output'transaction;
    count := count + 1;
  end process;
end acct_behavior;
```

Abbildung 3: Prozeßmodifizierende Vererbung in VistaTech OO-VHDL

5 Objekt-orientierte Methoden im HW-nahen SW-Bereich

Nachdem sich die OO-Methoden beim Entwurf "großer" Software bewährt haben, liegt es nahe, ihre Anwendbarkeit auch bei der Lösung "kleiner", HW-naher Probleme (z.B. Steuerungssysteme) zu untersuchen. Ein Beispiel findet sich in [Marti95], in dem die Steuerungssoftware einer Kaffeemaschine zuerst mit der Entwurfsmethode nach Booch [Booc94] entworfen und anschließend in C++ realisiert wird. Die bei diesem Experiment erreichte Codegröße von ca. 8KB liegt jedoch noch immer über dem auf preiswerten Microcontrollern verfügbaren ROM-Speicherplatz. Abhilfe brächte hier möglicherweise die Realisierung in einer kompakteren Sprache, z.B. Object-Assembler [Appl88].

Auch bei dem von uns bearbeiteten Problem (Abschnitt 6) wurde deutlich, daß selbst ein OO-Entwurf, bei dem auf Speichereffizienz geachtet wurde, die Ressourcen von wirtschaftlichen Kleinstmikrocontrollern sprengt. Auch die aktuellen Entwicklungen in Richtung OO- spezialisierter Embedded-Prozessorkerne (SUN picoJava), die dann eine direkte Ausführung von OO-Programmen erlauben würden, sind vermutlich für das Niedrigpreissegment noch zu unwirtschaftlich und daher impraktikabel.

Bei Vorhandensein von mehr Ressourcen (32-64 KB RAM) ist eine OO-Vorgehensweise aber auch im Steuerungsbereich sinnvoll. Hier dürften die durch leichtere Wartbarkeit und die Wiederverwendbarkeit gerade komplizierterer Steuerungsmodule erreichten Vorteile einen relativ betrachtet leicht erhöhten HW-Aufwand rechtfertigen.

6 EHS

Das von uns untersuchte Fallbeispiel ist eine hybride HW/SW-Applikation, die am ehesten dem Bereich Embedded Control zugeordnet werden kann. Der Schwerpunkt der Modellierung liegt dabei eindeutig auf der SW-Seite. Ursprünglich war auch eine Untersuchung der HW-Modellierbarkeit geplant, diese kam dann aber wegen der geringen Teilnehmerzahl nicht zustande.

Das an der Abt. E.I.S. durchgeführte Projektseminar behandelte Version 1.2 des "European Home System"-Standard EHS [Kung95]: EHS überdeckt den Bereich der Eigenheimautomatisierung und beschreibt das Verhalten

```
Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

-- ***********************************************************
-- generic counter
-- ***********************************************************

entity count8 is
        port (CLOCK: in STD_LOGIC;
              COUT: out STD_LOGIC_VECTOR (7 downto 0));
end count;

architecture count8_beh of count8 is
signal QOUT: STD_LOGIC_VECTOR (7 downto 0);
begin
  pcount : process (CLOCK)
    if (CLOCK'event and CLOCK='1') then  -- perform counting operation
              QOUT <= QOUT + "00000001";
    end if;
  end process;
COUT <= QOUT;
end count8_beh;

-- ***********************************************************
-- counter with asynchronous reset
-- ***********************************************************

entity count8_r is new count8
        port (CLEAR: in STD_LOGIC);
end count;

architecture count8_r_beh of count8_r is new count8_beh
begin
  pcount : process (CLEAR)    -- implicitly extended with CLOCK (use of super)
    begin
      if (CLEAR = '1') then   -- add reset behavior
            QOUT <= "00000000";
      else
            super pcount;     -- keep inherited counting behavior
      end if;
    end process;
end count8_r_beh;

-- ***********************************************************
-- counter with clock enable
-- ***********************************************************

entity count8_e is new count8
        port (ENABLE: in STD_LOGIC);
end count;

architecture count8_e_beh of count8_e is new count8_beh
begin
  pcount : process (ENABLE) -- implicitly extended with CLOCK (use of super)
    if (ENABLE = '1') then  -- add enable control
           super pcount;    -- keep inherited counting behavior
    end if;
  end process;
end count8_e_beh;
```

Abbildung 4: Vorschlag zur Einbettung von Prozessen der Oberklasse durch `super`

```
Library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

-- ************************************************************
-- counter with asynchronous reset flattened into standard VHDL
-- ************************************************************

entity count8_r is
        port (CLOCK, CLEAR: in std_logic;
              COUT: out std_logic_vector (7 downto 0) );
end count8_r;

architecture count8_r_beh of count8_r is new count8_beh
begin
  pcount : process (CLOCK, CLEAR)    --  extended with CLOCK (use of super)
    begin
      if (CLEAR = '1') then   -- reset behavior
            QOUT <= "0";
      elsif (CLOCK'event and CLOCK='1') then  -- perform counting operation
                  QOUT <= QOUT + "1";
      end if;
    end process;
end count8_r_beh;
```

Abbildung 5: Übersetzung von `count8_r` in Standard-VHDL

vielfältiger elektrischer Geräte sowie ihre Kommunikation untereinander und mit übergeordneten Steuerungseinheiten. Es definiert neben einer Applikationsschnittstelle für die Geräte auch Verwaltungs- und Netzwerkdienste, die die Einzelteile zu einem Gesamtsystem integrieren. Ein vertiefter Einstieg in das System wird erleichtert durch an der Abteilung bereits vorhandene praktische Erfahrungen mit EHS [Telk97].

Ziel des Projekts war, auf Grundlage der "flachen", eher klassisch- prozedural aufgebauten EHS-Spezifikation ein objekt-orientiertes Modell von EHS-Teilen und ihrer Abläufe aufzubauen. Dieses Modell könnte dann als Ausgangspunkt für eine weitergehende HW- Implementierung der Komponenten bzw. die Entwicklung mächtigerer Rapid-Prototyping- und Simulationswerkzeuge verwendet werden.

7 Vorgehen und Werkzeuge

Als Analyse- und Entwurfstechnik wurde die objekt-orientierten Methode von Booch [Booc94] eingesetzt. Wie viele aktuelle Beschreibungsverfahren hat auch diese eine graphikintensive Notation und ist ohne Einsatz geeigneter Hilfsmittel nur unkomfortabel zu verwenden. Die Teilnehmer verwendeten das einfache CASE-Werkzeug ObjectDomain zur Bearbeitung ihrer Modelle. Obwohl dessen Leistungsumfang im Vergleich zu etablierten kommerziellen Tools eingeschränkt ist (keine automatischen Konsistenzprüfungen, Eingabe nur in Booch-Notation ohne Jacobson-Konstrukte etc.), ermöglichte es doch ein flüssiges Arbeiten mit der Methode.

8 Modelle

Die so entstandenen Modelle gliedern sich in drei Schichten: Eine Applikationssicht setzt auf einer Kommunikationsschicht auf, die wiederum auf einer Simulationsschicht aufbaut. Im Fall einer realen HW-Implementierung würde die Simulationsschicht durch ein physikalisches Netzwerk-Interface ersetzt. Das Modell sieht aber bereits einen simultanen Betrieb von simulierten zusammen mit realen Geräten vor.

8.1 Applikationsschicht

Die OO-Applikationsschicht weist drei wesentliche Neuerungen gegenüber der Urspezifikation auf. Durch sogenannte *Proxy-Objekte* findet eine Kommunikation über das Netz völlig transparent statt. Aus Sicht des Anwendungsentwicklers wird lediglich eine Methode eines lokalen Objektes aufgerufen. Die Umsetzung in Telegramme und deren Adressieren und Verschicken geschehen unsichtbar und ohne Benutzerintervention. Auch das Zusammensetzen der Nachrichten entsprechend der EHS-Spezifikation geschieht transparent: Ein tabellenbasierter Umsetzungsmechanismus übersetzt den lokalen Methodenaufruf und seine aktuellen Parameter in eine entsprechende EHS-Nachricht. Der gleiche Mechanismus wird auch beim Auseinandernehmen einer eingetroffenen EHS-Nachricht verwendet, die so wieder in einen entsprechend parametrisierten lokalen Methodenaufruf umgewandelt wird. Die unterschiedlichen EHS-Konventionen für Anmeldung im Netz (enrolment), Zugriffskontrolle (tokenpassing) und die Bearbeitung der EHS-Kommandosprache (command language) lassen sich gut durch Vererbung abstrahieren. Abbildung 6 zeigt die Modellierung einer EHS-gesteuerten Lampe in der von uns vorgeschlagenen Form. Dabei braucht lediglich das Lampen-spezifische Verhalten in `Lamp_Interface` neu implementiert zu werden. Nachdem die o.g. Umsetzungstabelle noch um die vier Einträge erweitert wurde, die das EHS-Anwendungsfeld (application area) Beleuchtung in lokale Methoden und Aufrufparameter umsetzen, können sich Lampen-Objekte jetzt im EHS-Netz anmelden, Nachrichten austauschen und Zugriffe priorisiert bearbeiten. Diese Funktionalität wurde ohne Zutun des Anwenders durch Vererbung wiederverwendet.

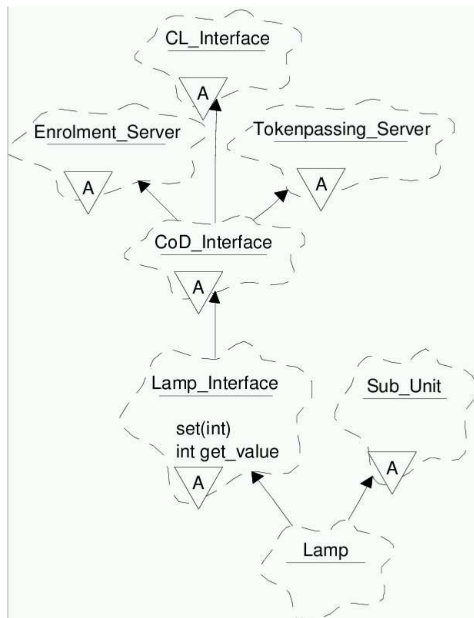

Abbildung 6: Modellierung einer EHS-Lampe

8.2 Kommunikationsschicht

Hier wurde der EHS-Protokollstapel modelliert. Dabei wurde insbesondere auf effizienten Umgang mit SpeicherRessourcen geachtet. So werden beispielsweise Pakete zwischen den Protokollschichten nicht umkopiert, sondern auf dem Weg in tiefere Schichten, die jeweils als Spezialisierung der darüberliegenden Schicht modelliert sind, um neue Attribute und Operationen ergänzt. Auf Schicht 2 (datalink/MAC) ist das Paket schließlich soweit ergänzt und spezialisiert, daß es auf das entsprechende Medium geschickt werden kann. Umgekehrt wird ein empfangenes Paket beim Aufstieg durch die Schichten wieder soweit generalisiert, daß die reine EHS-Nachricht zur Umsetzung ausgelesen werden kann. Ein ähnliches Vorgehen ist zwar auch bei der konventionellen Realisierung von Protokollstapeln möglich, dabei verschwimmen allerdings häufig die Grenzen zwischen den Protokollschichten (weniger übersichtlich, erhöhter Wartungsaufwand).

8.3 Simulationsschicht

Während Applikations- und Kommunikationsschicht auch in realen EHS-Geräten existieren, ist die Simulationsschicht spezifisch für den in Abschnitt 6 gewünschten Simulator. Da der Simulator nicht unter der Einschränkung von realen HW-Ressourcen entworfen werden mußte, konnten hier OO-Techniken auf breiter Front eingesetzt werden. So finden sich diverse der in [GHJV95] vorgestellten Entwurfsmuster wieder: Zur Überwachung von EHS-Objekten (monitoring) wird das *Observer*-Muster verwendet, der Aufbau komplizierter Auslöser (trigger) und der von ihnen ausgelösten Aktionen (stimulation) geschieht durch nach dem *Composite*-Muster entworfene Strukturen, und die Zuordnung von medien-spezifischen (powerline, twisted pair, radio frequency, infra red) MAC-Schichten an die modellierten Geräte wird mittels *Abstract Factory*-Objekten vorgenommen. Der so entstandene Entwurf ist außerordentlich mächtig und trotzdem leicht pfleg- und erweiterbar.

8.4 Vergleich konventionelles vs. OO-Vorgehen

Die bei konventionellem und OO-Vorgehen enstandenen Entwürfe K-EHS und OO-EHS für den EHS-Simulator unterscheiden sich deutlich. Während der K-Entwurf sich nur auf die Simulation (allerdings auch auf Co-Simulation von realen und simulierten Geräten) konzentriert, sind beim OO-Entwurf die Applikationsschicht und die Kommunikationsschicht durchaus auch in realen Geräten einsetzbar (mit den in Abschnitt 5 skizzierten Einschränkungen). Die Applikationsschicht des OO-Entwurfes mit ihrer Kapselung der EHS-Spezifika durch geeignete Abstraktionen fehlt im K-EHS völlig. Die Kommunikationsschichten sind, bedingt durch die Erfüllung der gleichen Protokollbeschreibung, relativ ähnlich. Das nicht auf HW ausgerichtete K-EHS legt aber bei weitem nicht das Gewicht auf Speicherkonservierung wie OO-EHS. Die Simulationsschichten sind fast völlig verschieden, da (wie in Abschnitt 8.3 beschrieben) gerade dieser Teil von OO-EHS unter konsequenter Anwendung von OO-Techniken entworfen wurde. Die Mächtigkeit der OO-EHS Simulatorschicht liegt auch daher wesentlich über der von K-EHS.

Bei der Erstellung der beiden Entwürfe zeigte sich, daß das klassische Vorgehen wesentlich eher zu Prototypen führte. Von K-EHS existiert eine funktionsfähige Implementierung, während OO-EHS aus Zeitmangel nur die Entwurfsphase durchlaufen hat. Das inkrementelle und iterative Vorgehen, das typisch für OO-Methoden ist, kostet sehr viel Zeit und kommt anfangs auch nur langsam in Gang. Mit wachsender Erfahrung beim Umgang mit den ersten Modellen erhöht sich die Produktivität dann aber wesentlich. Im Fall des OO-EHS Projektseminars wurden die größten Fortschritte erst wenige Wochen vor Ende des des Projektes erreicht. Es wäre nun interessant gewesen zu untersuchen, in wieweit der wesentlich erhöhte Entwurfsaufwand nicht nur zu einem mächtigeren und vielseitigeren Modell als K-EHS geführt hat, sondern auch zu einer Erleichterung der anschließenden Implementierung hätte beitragen können. Aus Personal- und Zeitmangel war diese im Rahmen des Projektseminars aber nicht mehr durchführbar. Nach praktischen Erfahrungen [Booc94] liegt der Entwurfsaufwand bei OO-Vorgehen bei rund 166% des bei K-Vorgehens üblichen, amortisiert sich aber durch einen auf 60% reduzierten Implementierungsaufwand, auf 75% reduzierten Test-, und auf 30% reduzierten Integrationsaufwand.

9 Zusammenfassung

Durch geeignete Erweiterung von HDLs, gezeigt am Beispiel von OO-VHDL, können OO-Entwurfsmethoden direkt auf den VLSI-Entwurf angewandt werden. Trotz der Einschränkungen, die im wesentlichen auf den statischen Charakter einer gefertigten Schaltung zurückzuführen sind, kann eine Verbesserung in bezug auf Pfleg- und Wiederverwendbarkeit erreicht werden.

Die Erstellung von HW-naher Software mit OO-Methoden ist, soweit minimale Ressourcen nicht das primäre Optimierungsziel darstellen, ebenfalls sinnvoll. Auch hier kann durch die gesteigerte Ausdrucksfähigkeit ein Modell entwickelt werden, das sich flexibel an Änderungen anpassen läßt und eine weitgehende Wiederverwendung von Komponenten erlaubt.

Literatur

[AgMS95] Agsteiner, K., Monjau, D., Schulze, S., “Object-Oriented High-Level Modeling of System Components for the Generation of VHDL Code”, Proc. EURODAC 1995, Brighton, UK, 1995

[Appl88] Apple Computer, Inc., “Macintosh Programmers Workshop Assembler, Appendix H: Object Assembler Macros”, Cupertino, USA, 1988

[Booc94] Booch, G, “Object-Oriented Analysis and Design, 2nd ed.”, Addison-Wesley 1994

[Cox86] Cox, B.J., “Object Oriented Programming”, Addison-Wesley 1986

[DiMo92] Dion, J., Monier, L., “Design Tools for BIPS-0”, DEC WRL Tech Note-32, Dec. 1992

[HWSW91] Howard, D., Walczowski, L.T., Smith, M.H., Waller, W.A.J., “C.S.L.: A Language for Process Independent V.L.S.I. Design”, IEE Colloquium on Object-Oriented Programming, Feb. 1991

[JCJO92] Jacobson, I., et al., “Object-Oriented Software Engineering”, Addison-Wesley 1992

[GHJV95] Gamma, E., Helm, R., Johnson, R., Vlissides, J., “Design Patterns”, Addison-Wesley 1995

[Kung95] Kung, A., et al., “The European Home Systems Network”, Trialog 1995

[Marti95] Martin. R.M., “Designing Object-Oriented Applications using the Booch Method”, Prentice-Hall 1995

[MoDi95] Monier, L., Dion, J., “Recursive Layout Generation”, DEC WRL Research Report 95/2, March 1995

[Quib95] Quibeldey-Cirkel, “Das Objekt-Paradigma in der Informatik”, Teubner 1995

[Telk97] Telkamp, G., “Entwurf von Home Automation Systemen in der Ausbildung”, eingereicht zum E.I.S. Workshop 1997

[Vist96] Vistatech, Inc., “Inheritance in OO-VHDL”, http://www.vistatech.com

[WWNS91] Walczowski, L.T., Waller, W.A.J., Nalbantis, D., Shi, K., “Analog VLSI Synthesis – A Fast Approach”, Electronic Engineering Labs., U. of Kent, Canterbury, UK, 1991

Dank den Teilnehmern des OO-EHS Projektseminars im WS 96/97, Stefan Hollesch, Kerstin Reese und Carsten Warnk, für intensive Mitarbeit und fruchtbare Diskussion.