Synthese und Simulation für mehrere Zielsprachen

Dipl.-Inform. Dirk Herrmann und Prof. Dr. Rolf Ernst
Institut für Datenverarbeitungsanlagen, Technische Universität Braunschweig

Kurzfassung

Die Erzeugung einer Schaltungsbeschreibung auf Register-Transfer-Ebene in einer Hardwarebeschreibungssprache ist der letzte Schritt der High-Level Synthese. Wir stellen einen allgemeinen Ansatz zur Ausgabe von Register-Transfer-Beschreibungen vor, der die Unterstützung mehrerer Zielsprachen auf einheitliche Weise erlaubt, ebenso wie eine nachfolgende zyklentreue Simulation der Schaltung. Erreicht wird dies durch die Einführung einer Abstraktionsschicht zwischen den Datenstrukturen des High-Level Synthesesystems und der Syntax der Zielsprachen sowie durch die Verwendung eines gemeinsamen Simulationsausgabeformates. Es werden Ergebnisse für die Implementierungen der Zielsprachen VHDL, Verilog und C++ vorgestellt.

1 Einführung

Als High-Level Synthese bezeichnet man die Transformation der Verhaltensbeschreibung einer Applikation in eine Beschreibung auf der Register-Transfer-Ebene (*RT-Ebene*) [4]. Als Zielsprache dient eine Hardwarebeschreibungssprache, die durch Register-Transfer-Synthesewerkzeuge weiterverarbeitet werden kann. Von kommerziellen Synthesesystemen wird erwartet, daß dem Schaltungsentwickler mehrere Zielsprachen zur Auswahl stehen. Für die Entwickler der Synthesesysteme bedeutet allerdings die Unterstützung jeder zusätzlichen Ausgabesprache eine Erhöhung des Entwicklungs- und Wartungsaufwandes. Unseres Wissens ist die effiziente Erzeugung von RT-Beschreibungen („Netzlisten") für mehrere Zielsprachen bisher nicht untersucht worden.

Im folgenden wird ein Ansatz für die abschließende Netzlistengenerierungsphase der High-Level Synthese und eine darauffolgende taktzyklentreue Simulation vorgestellt. Durch eine zusätzliche Abstraktionsschicht für die Netzlistengenerierung erlaubt unser Ansatz die generische Erzeugung von Code für mehrere Zielsprachen, ohne die Qualität der synthetisierten Schaltungen zu beeinflussen. Der benötigte Programmcode zur Implementierung der Ausgabe in einer Zielsprache ist gering im Verhältnis zu dem für alle Sprachen gemeinsam genutzten Code (ein Fünftel oder weniger), so daß der Wartungsaufwand des High-Level Synthesesystems gering bleibt. Weiterhin wird durch die Nutzung von gemeinsamem Programmcode erreicht, daß der Aufbau des generierten Codes für alle Zielsprachen ähnlich ist, was die Fehlersuche vereinfacht.

Die RT-Beschreibungen enthalten Ausgabeanweisungen, die während der Simulation den Schaltungszustand wiedergeben, jedoch bei der Synthese ignoriert werden können. Es wird ein einheitliches, an Lisp angelehntes Ausgabeformat zur Wiedergabe des Simulationsablaufes verwendet. Eine Nachbearbeitung der Simulationsausgaben liefert eine zyklenbasierte Darstellung des Ablaufs und der Resultate der Simulation. Da hierbei Unterschiede zwischen den verschiedenen Zielsprachen (wie beispielsweise die Prozeßausführungsreihenfolge bei VHDL und Verilog) kompensiert werden, zeigt die so erhaltene Darstellung einen für alle Zielsprachen identischen Simulationsverlauf. Da auch die Simulationsstimuli aus einem gemeinsamen Eingabeformat erzeugt werden, ist der gesamte Prozeß der Netzlistengenerierung und anschließenden Simulation zielsprachen- und simulatorunabhängig.

Die Netzlistenerzeugung wurde in C++ implementiert und erlaubt die Generierung von Netzlisten in VHDL [3], Verilog [11] und C++ [10]. Für die Zielsprache C++ wurde zur Modellierung paralleler Ausführung von Prozessen eine zusätzliche Laufzeitbibliothek bereitgestellt. Eine zyklentreue Simulation der Schaltungen ist in allen Zielsprachen möglich, allerdings zeigte sich, daß die Kompilation des erzeugten C++ Codes sehr zeitaufwendig ist. Die Nachbearbeitung der Simulationsausgaben erfolgt in Scheme, einem Dialekt von Lisp. Durch das Simulationsausgabeformat in Lisp Syntax gestaltet sich der Code zur Nachbearbeitung in Scheme äußerst einfach und erlaubt, während der Simulation beliebig komplexe Bedingungen zu überprüfen.

Im nächsten Abschnitt geben wir einen Überblick über den Gesamtablauf in unserem Synthesesystem. Der Abschnitt 3 beschreibt die weitgehend zielsprachenunabhängige Netzlistengenerierung. Im Abschnitt 4 wird die zielsprachen- und simulatorunabhängige Simulati-

Diese Arbeit wird von der Deutschen Forschungsgemeinschaft (DFG) unterstützt.

on vorgestellt. Es folgen ein Abschnitt mit einigen Resultaten und schließlich eine Zusammenfassung.

2 Ablauf der Synthese

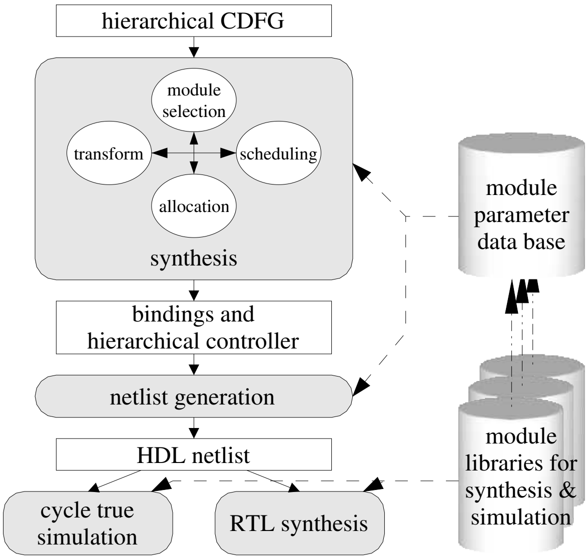

Abbildung 1: Ablauf der High-Level Synthese

Die **Abbildung 1** zeigt den Ablauf des Syntheseprozesses im Braunschweiger Synthese System (*BSS*), das für unsere Experimente verwendet wurde [5]. Die Eingabe bildet ein hierarchischer Kontroll- und Datenflußgraph (*CDFG*), d. h. jeder Knoten des Graphen ist entweder eine Operation oder wiederum ein CDFG. Durch die Hierarchie werden Kontrollstrukturen wie geschachtelte Schleifen oder Funktionsaufrufe repräsentiert. Die Teilschritte Codetransformation, Modulauswahl, Scheduling und Allokation werden iterativ auf die Eingabebeschreibung angewendet. Als Resultat des Synthesevorgangs ergibt sich eine Abbildung von Operationen auf funktionale Einheiten (*FUs*) sowie ein hierarchischer Steuergraph, der die Struktur des CDFG wiedergibt nachdem sämtliche Transformationen ausgeführt wurden.

In **Abbildung 2** wird ein vereinfachtes Beispiel eines Steuergraphen mit zwei Hierarchieebenen gezeigt sowie die entsprechenden CDFGs. Der obere Teil der Abbildung 2 zeigt links den übergeordneten Steuergraphen und rechts den dazugehörigen CDFG. In diesem Beispiel wird ein Wert aus dem Speicher gelesen und als Parameter an den untergeordneten CDFG übergeben. Dieser liefert einen Rückgabewert, der in den Speicher zurückgeschrieben wird. Der untere Teil der Abbildung zeigt grau unterlegt den untergeordneten Steuergraphen und CDFG. Diese bilden eine Schleife, in der der Eingabewert inkrementiert wird bis eine bestimmte Abbruchbedingung erfüllt ist. Die Steuergra-

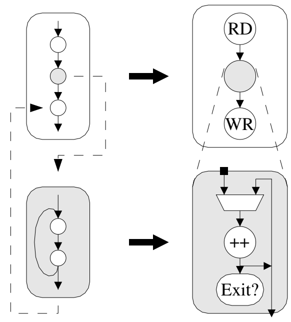

Abbildung 2: Hierarchie des Steuergraphen

phen kommunizieren über Start- und Stopsignale um sich gegenseitig zu aktivieren bzw. über Beendigung zu informieren. Abbildung 2 soll lediglich einen Einblick in den Aufbau des Syntheseergebnisses geben, weshalb Details wie Registerzugriffe nicht dargestellt sind und auch die Tatsache unberücksichtigt ist, daß der Steuerpfad eine Pipelinestruktur aufweist.

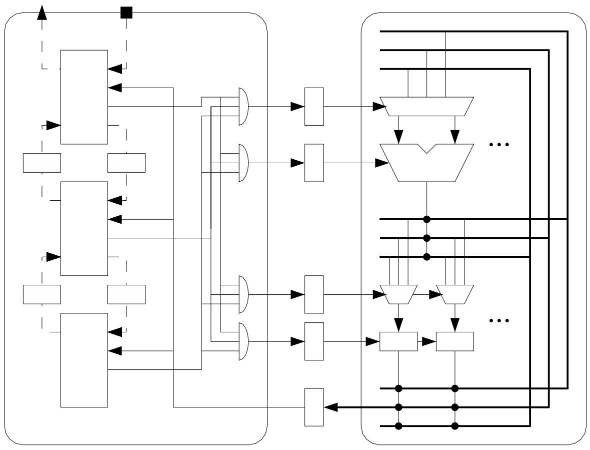

Abbildung 3: Syntheseergebnis

Die Register-Transfer-Struktur der Schaltung, wie sie sich durch den Steuergraphen und die Zuordnung von Operationen zu FUs ergibt, ist in **Abbildung 3** mit dem Steuerpfad auf der linken Seite und dem Datenpfad rechts dargestellt. Für jeden Steuergraphen der Hierarchie wird ein eigener endlicher Steuerautomat, ein *Steuerblock*, erzeugt. Die Ausgangssignale des Steuerpfades werden aus den Ausgangssignalen der einzelnen Steuerblöcke durch eine einfache ODER-Verschaltung erzeugt. Alle Ein- und Ausgangssignale des Steuerpfades sowie die Signale zur Kommunikation zwischen den Steuerblöcken werden über Register geführt, um den kritischen Pfad zu reduzieren. Innerhalb des Datenpfads erfolgt der Datentransfer über ein spezialisiertes Verbindungsnetzwerk, wobei auch direkt verbundene FUs (*chaining*) oder Schieberegister zulässig sind.

Alle Syntheseschritte und die nachfolgende Netzli-

stengenerierung verwenden eine gemeinsame Datenbasis mit Informationen über die verfügbaren Typen von FUs. Für jede Art von FU beschreibt die Datenbasis die ausführbaren Operationen, Ausführungszeiten, Flächenbedarf und auf welche Weise die Ein- und Ausgänge anzuschließen sind. Die Datenbasis enthält folglich nur Modellparameter und Anschlußinformationen, und zwar in einer Form, die unabhängig von der Zielsprache und der Implementierung der FUs ist. Für eine nachfolgende RT-Synthese und Simulation müssen jedoch Bibliotheken mit der Implementierung der FUs in den jeweiligen Sprachen existieren.
Im folgenden unterscheiden wir zwischen dem *Zeitraum der Netzlistengenerierung*, in dem die RT-Beschreibung in der ausgewählten Zielsprache erzeugt wird, und der *Synthese*-, *Simulations-* oder *Laufzeit*, in der eine bereits erzeugte RT-Beschreibung synthetisiert, simuliert oder (im Fall von C++) ausgeführt wird.

3 Erzeugung der Netzliste

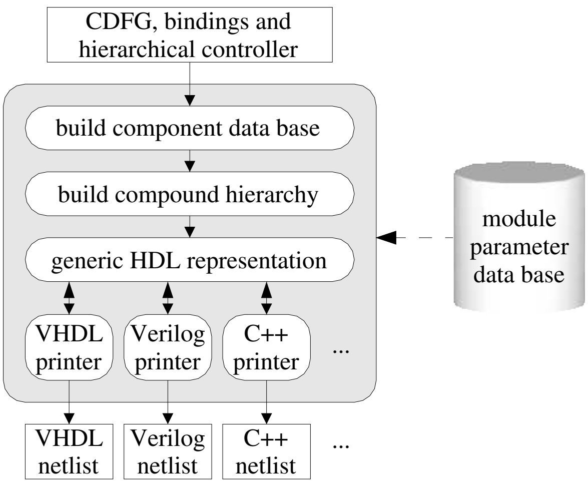

Abbildung 4: Ablauf der Netzlistenerzeugung

Abbildung 4 zeigt die unterschiedlichen Phasen der Netzlistenerzeugung. In der ersten Phase wird eine Datenbank von Komponenten angelegt, die erzeugt werden sollen wie z. B. Instantiierungen von FUs, Register, Multiplexer und Steuerblöcke.
In der zweiten Phase werden unterschiedliche Komponenten hierarchisch gruppiert, wie in **Abbildung 5** gezeigt ist. Im Gegensatz zu einem flachen Design erlaubt dies eine inkrementelle bottom-up RT-Synthese. Die Komponentenhierarchie ist frei zusammenstellbar, d. h. es sind andere Konstellationen als in Abbildung 5 möglich. Beispielsweise lassen sich Teile des Kontroll- und Datenpfades als eigene Komponente zusammenfassen, sofern dies vorteilhaft erscheint. Die Simulationsumgebung, also externe Speicher und die Taktgenerierung, wird in einer übergeordneten Hierarchiestufe mit der Schaltung zusammengefaßt.

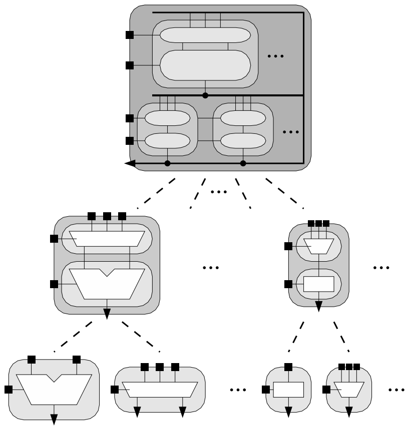

Abbildung 5: Komponentenhierarchie

Die dritte Phase bereitet die eigentliche Ausgabe vor. Die Detailstruktur jeder Komponente, d. h. die Prozesse, Signalzuweisungen und Ausdrücke, werden in einem internen Format dargestellt, das eine Abstraktion der Zielsprachen repräsentiert. Die folgende abschließende Ausgabephase kann dieses Format leicht in die Syntax der gewählten Zielsprache umsetzen.

3.1 Abstraktion von der Zielsprache

Nach der zweiten Phase der Netzlistengenerierung liegt bereits eine interne Darstellung der Schaltung vor, die alle notwendigen Details enthält, um eine Ausgabe in einer beliebigen RT-Beschreibungssprache zu ermöglichen. Allerdings liegen die meisten Informationen nur implizit vor, codiert in Form von internen Datenstrukturen des Synthesesystems. Die Bedingungen für Zustandsfortschaltungen innerhalb der Steuerblöcke beispielsweise sind durch die Struktur der Steuergraphen (siehe Abbildung 2) definiert. Durch die Einführung einer Abstraktionsschicht zwischen der impliziten Schaltungsdarstellung und der Darstellung in der Syntax der Zielsprache ist es uns gelungen, den Großteil des zielsprachenspezifischen Programmcodes durch generische Routinen zu ersetzen.
Ausgehend von VHDL und Verilog als Zielsprachen erlaubt unsere interne Darstellung die Verwendung von Signalen und Registern unterschiedlicher Bitbreiten, `if-then-else` und `case` als Kontrollstrukturen sowie boolesche Ausdrücke mit Operatoren wie beispielsweise UND, ODER, NICHT, Vergleich, Konkatenation und Bitauswahl. Die Berechnungen erfolgen in parallelen Prozessen, die durch Ereignisse, d. h. Änderungen von Signalwerten, angestoßen werden. Hierarchisch schachtelbare Module dienen zur Aufteilung der Schaltung in separat synthetisierbare Komponenten.
Dieses Zwischenformat erlaubt eine einfache

Umsetzung nach VHDL und Verilog, wobei RT-Beschreibungen auf einer hohen Abstraktionsebene erzeugt werden. Die Zustandsfortschaltung der Steuerblöcke wird beispielsweise als `case`-Anweisung ausgegeben. Sowohl bei der Erzeugung des Zwischenformates als auch bei der Umsetzung in die Zielsprachen wurde sichergestellt, daß die Qualität der Schaltung nach der RT-Synthese (in unserem Fall durch Synopsys) unverändert blieb.
Neben der Codeerzeugung für VHDL und Verilog wurde auch die Ausgabe nach C++ mit dem Ziel implementiert, schnelle Simulationsläufe zu ermöglichen. Um vergleichbare Simulationsergebnisse zu erhalten, wurde C++ in Form einer Laufzeitbibliothek um Datenstrukturen für Prozesse und Signale ergänzt. Weiterhin wären Sprachen wie hardwareC [7] oder TH [8] leicht zu unterstützen. Für Formate wie SLIF (kombinatorische Logik) oder KISS (tabellarische Automatendarstellung), ist die interne Darstellung nur eingeschränkt geeignet, da diese Zielsprachen nur Operationen auf Signalen der Bitbreite eins erlauben und keine Kontrollstrukturen zur Verfügung stellen [7].
Das interne Zwischenformat erlaubte die Vereinheitlichung großer Teile der Ausgabeerzeugung. Lediglich für die Umsetzung dieses Formates in die jeweiligen Zielsprachen mußten spezielle Codeerzeugungsroutinen entwickelt werden. Durch diese spezialisierten Routinen konnten neben den syntaktischen Unterschieden auch konzeptionelle Differenzen zwischen den Zielsprachen behandelt werden. Als Beispiel sei die Verbindungsstruktur genannt, die sich in VHDL und Verilog implizit durch die Deklaration und das Anschließen von Signalen ergibt. In C++ müssen diese Verbindungen explizit erzeugt werden. Wie im folgenden beschrieben wird, bestanden die sprachspezifischen Codeanteile jedoch typischerweise nur aus kurzen Codestücken.

3.1.1 Signalnamen

Signalnamen werden durch einen zentralen Algorithmus vergeben. Eine an die Zielsprache angepaßte Nachbearbeitung garantiert, daß die Namen keine Schlüsselwörter darstellen oder unerlaubte Zeichen enthalten. Ein konsistentes Vorgehen bei der automatischen Vergabe von Signalnamen hat einen wesentlichen Einfluß auf die Verständlichkeit der generierten RT-Beschreibungen. Daher wird ein Schema verwendet, das es erlaubt, aus dem Namen des Signals auf die Funktion innerhalb der Schaltung zurückzuschließen: Steuersignale werden anhand der Zielkomponente bezeichnet während Datensignale nach der Quellkomponente benannt werden. Zusätzlich werden Informationen über die Art des Anschlusses innerhalb der aktuellen Komponente aufgenommen, z. B. ob es sich um ein eingehendes, ausgehendes oder lokales Signal handelt.

3.1.2 Ausdrücke

Signalzuweisungen oder Bedingungen für Zustandsübergänge werden in unserem Zwischenformat als boolesche Ausdrücke dargestellt. Diese Ausdrücke werden intern als Bäume repräsentiert, ähnlich den *parse trees* im Compilerbau [2]. Blattknoten sind Konstanten oder Signale, alle anderen Knoten sind Operationen wie Vergleich, logisches oder bitweises UND, ODER oder NICHT. Durch die Baumstruktur ist es unnötig, Bindungsstärken für die Operatoren festzulegen. Bei der Ausgabe der Ausdrücke wird durch Klammerung die gewünschte Operatorbindung in der jeweiligen Zielsprache erzwungen.

```
printLogicOr(Lang lg, Node l, Node r) {
    printLeftParen(lg);
    printNode(lg, l);
    if ((lg == CPP) || (lg == VERILOG))
        print(" || ");
    else if (lg == VHDL)
        print(" or ");
    else illegalLanguageError();
    printNode(lg, r);
    printRightParen(lg);
}
```

Abbildung 6: Rekursive Ausgabe von Ausdrücken

Die Ausgabe in die Zielsprache erfolgt rekursiv, wie es vereinfacht in **Abbildung 6** anhand der Ausgabe eines logischen ODER dargestellt ist. In dem Beispiel dient die Funktion `printNode` dazu, den als Parameter übergebenen Knoten auszugeben, indem anhand dessen Knotentyps in die entsprechende Ausgabefunktion verzweigt wird. Das Beispiel zeigt, wie gering der zielsprachenspezifische Anteil des Codes sein kann. Mit Hilfe dieses Ansatzes lassen sich Ausdrücke wie `(((state == 1) or (state == 5)) and ((s1 = 0) or ((s1 = 2) and (s2 = 0))))`, die innerhalb der Steuerblöcke auftreten können, in einer zentralen Routine vorbereiten, die für alle Zielsprachen genutzt wird.

3.1.3 Kontrollanweisungen

Zur Erzeugung von RT-Beschreibungen sind nur wenige Kontrollstrukturen notwendig, wie z. B. bedingte Ausführung oder Prozesse. Schleifenanweisungen werden dagegen nicht benötigt. Mit der Möglichkeit, Ausdrücke generisch zu behandeln, lassen sich `if-then-else` Anweisungen einfach implementieren: Ein Ausdruck stellt die zu überprüfende Bedingung dar, und die `then` und `else` Blöcke bestehen aus Sequenzen von Zuweisungsausdrücken.
Die Ausgabe von Prozessen machte wegen der größeren Syntaxunterschiede mehr zielsprachenabhängigen Code nötig. Dennoch konnten bestimmte Teile, wie

z. B. die Ausgabe von Prozeßaktivierungslisten (*sensitivity lists*) generisch behandelt werden. Kontinuierliche Zuweisungen (*continuous assignments*) in VHDL und Verilog stellen eine Kurzform für Prozesse dar, die nur aus einer einzelnen Signalzuweisung mit impliziter Aktivierung bestehen. In C++ mußten diese jedoch wie normale Prozesse mit expliziter Aktivierung behandelt werden.

3.1.4 Module

Jede Komponente (Steuerblöcke, Multiplexer, Register, FUs und Gruppierungen von Komponenten) wird als separates Modul der Zielsprache in einer eigenen Ausgabedatei erzeugt. Obwohl VHDL und Verilog beide ein Modulkonzept haben, bestehen große Unterschiede: Für VHDL muß streng zwischen der Schnittstellendefinition (*entity*) und der Implementierung (*architecture*) unterschieden werden, während dies in Verilog nicht nötig ist. Aufgrund solcher Unterschiede konnten nur kleinere Teile der Ausgabefunktionen für Module generisch realisiert werden, wie beispielsweise das Erzeugen der Parameterliste der Modulschnittstellen.
Die Instantiierung von Modulen innerhalb anderer Module hingegen erfordert im wesentlichen das Ausgeben der Parameterliste. Im Falle von FUs (die in einer externen Bibliothek implementiert werden), werden diese Informationen aus der in Abschnitt 2 beschriebenen Datenbasis entnommen.

3.1.5 Simulationsspezifische Anweisungen

Um zu garantieren, daß sich die simulierte und synthetisierte Schaltung entsprechen, wird keine Unterscheidung zwischen Netzlistenerzeugung für Simulation und Synthese gemacht. Simulationsspezifische Ausgabeanweisungen werden immer in die RT-Beschreibung eingefügt, aber so, daß sie bei der Synthese ignoriert werden können ohne das Verhalten der Schaltung zu beeinflussen. Die Informationen, die von den Komponenten während der Simulation ausgegeben werden, sind für alle Zielsprachen gleich.

3.2 Implementierung

Die Implementierung der Netzlistengenerierung in C++ macht Gebrauch von Klassen, Ableitungshierarchie, Templates und spezialisierten Streamklassen. Auf Basis dieser von C++ gebotenen Möglichkeiten wurde die zielsprachenunabhängige Codeerzeugung entwickelt. Im folgenden wird skizziert, wie diese Konzepte von C++ bei der Implementierung genutzt wurden.

Ableitungshierarchie: Die verschiedenen Knotentypen der Ausdrücke (siehe Abschnitt 3.1.2) sind von einem gemeinsamen Basisknotentyp abgeleitet. Aus diesem Grund mußte die Funktion `printNode` in Abbildung 6 nicht tatsächlich implementiert werden, da der Mechanismus der virtuellen Funktionsaufrufe in C++ dies implizit realisiert. Durch die Verwendung der Ableitungshierarchie konnte die Menge der Knotentypen während der Implementierung erweitert werden, ohne bestehenden Code zu beeinflussen.

Templates: Templates in C++ erlauben, verschiedene Implementierungen derselben Funktion bereitzustellen, wobei die Auswahl bei der Kompilation erfolgt. Die Definition des Aufzählungstyps `Lang` mit den Elementen `VHDL`, `Verilog` und `CPP` und die Deklaration

```
template <Lang L> void printObj();
```

erlauben, eine Standardimplementierung der Funktion `printObj` sowie zielsprachenspezifische Implementierungen

```
template <> void printObj<VHDL>();
template <> void printObj<Verilog>();
template <> void printObj<CPP>();
```

für diejenigen Sprachen bereitzustellen, für die die Standardimplementierung nicht genügt. Der Vorteil dieses Ansatzes besteht in der Möglichkeit, spezielle Implementierungen selektiv einzufügen, da der Compiler automatisch erkennt, welche Spezialimplementierungen verfügbar sind und entsprechenden Code generiert.

Streamklassen: Die Verwendung zielsprachenspezifischer Streamklassen machte sehr viel syntaxabhängigen Code unnötig. Sei `s` ein Ausgabestream, der 'weiß' für welche Zielsprache Code erzeugt werden muß. Wenn der Stream `s` angewiesen wird, eine Zeichenkette als Kommentar auszugeben, wird die Zielsprachensyntax für Kommentare automatisch ergänzt.

4 Simulation

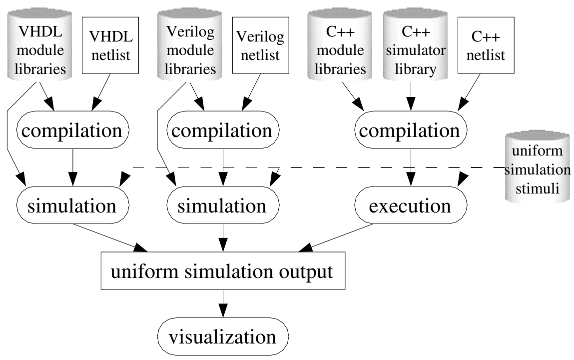

Abbildung 7: Ablauf der Simulation

Die Simulation der RT-Beschreibungen erfolgt mit den Simulatoren der jeweiligen Zielsprache (siehe **Abbildung 7**). Simulationsstimuli werden in einem einheitlichen Format in Lisp Syntax definiert, das automatisch in das für den jeweiligen Simulator geeignete Format übersetzt wird. Die RT-Beschreibungen melden den Zustand der Simulation in einem einheitlichen, ebenfalls an Lisp angelehnten Ausgabeformat. Unabhängig von der Zielsprache werden die Simulationsausgaben aufbereitet, um den Ablauf der Simulation zu visualisieren.
Für jede Zielsprache, für die eine Simulation gewünscht wird, müssen als zielsprachenabhängige Zusatzdateien lediglich die Bibliotheken mit den Implementierungen der FUs bereitgestellt werden. C++ benötigt zusätzlich die Laufzeitbibliothek mit den Implementierungen der Prozeß- und Signalklassen sowie eine Routine zur Realisierung eines ereignisgesteuerten Simulationsablaufes. Bei der Kompilation von C++ Netzlisten werden diese Bibliotheken in die ausführbare Datei eingebunden.

4.1 Generische Simulation

Um eine einheitliche zyklentreue Simulation unabhängig vom gewählten Simulator oder der Zielsprache zu ermöglichen, wurde ein generisches Ausgabeformat in Lisp Syntax definiert. Während der Simulation wird der gesamte Systemzustand ausgegeben (Taktzyklus, Zustandsübergänge, Signalwerte, Prozeßausführungen etc.). Funktionseinheiten aus der Simulationsbibliothek müssen entsprechend über ihren inneren Zustand berichten. Auf diese Weise können gemeinsame Werkzeuge zur Nachbearbeitung und Analyse der Simulationsausgaben verwendet werden. Außerdem ist die Zuordnung von Simulationsereignissen zu den Operationen des ursprünglichen CDFG leicht herzustellen. Dies wäre ohne die Verwendung eines eigens definierten Ausgabeformates schwierig zu bewerkstelligen, da jede Komponente der RT-Beschreibung einer großen Menge von Operationen des CDFG zugeordnet sein kann.

```
 ...
 (cycle 12)
 (state-change Block_7 "S1")
 (module-in "NOT_0" ((sig "I" 0)))
 (module-out "NOT_0" ((sig "O" 1)))
 (mux-in "MUX_5" ((sig "NOT_0_O" 1)))
 (mux-out "MUX_5" ((sig "REG_0_I" 1)))
 (module-in "REG_0" ((sig "I" 1)
                     (sig "WR" 1)))
...
```

Abbildung 8: Beispiel einer Simulationsausgabe

Das Ausgabeformat in Lisp Syntax, wie es in **Abbildung 8** dargestellt ist, wurde gewählt wegen seiner Einfachheit und um die Definition einer eigenen Simulationsausgabesyntax zu vermeiden. Da Lisp (oder in unserem Fall Scheme) es erlaubt, hierarchische Klammerstrukturen mit einem einzigen Kommando einzulesen, erübrigte sich die Notwendigkeit eines eigenen Parsers. Die Simulationsausgaben können mit einem beliebigen Lisp- oder Schemeinterpreter verarbeitet werden, was während der Fehlersuche erlaubt, die Simulationsausgaben als Datenstruktur einzulesen und mit sämtlichen Möglichkeiten einer Programmiersprache zu analysieren.
Die Nachbearbeitung der Simulationsausgaben ist aus drei Gründen notwendig. Erstens ist die Prozeßausführungsreihenfolge in VHDL und Verilog nicht deterministisch. Die Folge von Ereignissen kann daher zwischen den Sprachen variieren, aber auch zwischen verschiedenen Simulatoren für dieselbe Sprache und theoretisch sogar zwischen unterschiedlichen Simulationsläufen auf demselben Simulator. Zweitens kann innerhalb eines simulierten Taktzyklus auf dasselbe Signal mehrmals schreibend zugegriffen werden. Um die funktionale Korrektheit zu überprüfen ist jedoch nur der Wert des Signals zu Beginn des nächsten Taktes von Bedeutung. Drittens treten, durch die Pipelinestruktur des Kontrollpfades bedingt, zusammengehörige Ereignisse im Kontroll- und Datenpfad in unterschiedlichen Taktzyklen auf. Bei der Visualisierung des Simulationsablaufes bevorzugen wir jedoch eine Darstellung, bei der zusammengehörende Ereignisse zur verbesserten Lesbarkeit zusammen dargestellt werden. Daher wird eine (konfigurierbare) Umsortierung der Simulationsereignisse vor der Visualisierung vorgenommen.

5 Ergebnisse

	#Zeil.	% Zeil.	#Funk.	$\frac{\#Zeil.}{\#Funk.}$
gesamt	4498	100.00	497	9.1
generisch	2941	65.38	277	10.6
VHDL	587	13.05	75	7.8
Verilog	429	9.54	73	5.9
C++	541	12.03	72	7.5

Tabelle 1: Codezeilen und Funktionsanzahlen

Tabelle 1 zeigt die Anzahl von Codezeilen und Funktionen der Implementierung der Netzlistengenerierung. Die Zeilenanzahlen enthalten weder Leerzeilen noch Kommentarzeilen und auch keine Zeilen, die lediglich aus öffnenden oder schließenden geschweiften Klammern bestehen. Die Gesamtzahl von Zeilen und Funktionen ist in der ersten Zeile dargestellt. Alle weiteren Zeilen zeigen die absoluten und relativen Anteile des generischen Codes und des Codes für jede der Zielsprachen. Bezogen auf die Zeilenzahl des generischen

Codes benötigt Verilog lediglich ungefähr ein siebtel zusätzlicher Codezeilen und VHDL etwa ein fünftel, was den Nutzen und die Effizienz unseres Abstraktionsansatzes belegt.
Wie schon Abbildung 6 andeutet, sind viele der sprachspezifischen Funktionen trivial. Dies wird durch die Tatsache bestätigt, daß die durchschnittliche Zeilenzahl je spezialisierter Funktion (inklusive der Zeilen mit dem Funktionskopf) im Bereich von 5.9 bei Verilog bis 7.8 bei VHDL liegt, während die durchschnittliche Länge der generischen Funktionen 10.6 Zeilen beträgt. Da für jede Zielsprache einige Hilfsfunktionen benötigt wurden, variiert die Zahl der Funktionen pro Zielsprache leicht. Abgesehen davon zeigte sich, daß die meisten Funktionen entweder generisch implementierbar waren, oder aber für jede Zielsprache eine eigene Implementierung benötigten. Die unterschiedlichen Codezeilen bei den verschiedenen Zielsprachen deuten auf die syntaktische Komplexität der Sprachen hin [9, 6].
Bei C++ konnte die Komplexität dadurch begrenzt werden, daß die Laufzeitbibliothek mit dem Ziel entwickelt wurde, eine einfache Netzlistenerzeugung zu ermöglichen. Die Tatsache, daß C++ eine universelle Programmiersprache und keine Hardwarebeschreibungssprache ist, wirkte sich daher nicht aus. Auch das für C++ erforderliche explizite Herstellen der Signalverbindungen (siehe Abschnitt 3.1) hatte auf den Implementierungsaufwand keinen wesentlichen Einfluß. Diese Besonderheit ließ sich zentral durch eine einzelne Funktion erfassen.

	VHDL	Verilog	C++
bsearch	7191	6360	11359
bsort	8057	6762	12362
diffeq	8052	7121	12468
gcd	5907	5072	9108
ellipf	6977	6406	10582
calc	4142	3664	6544
max	5766	5145	8832
filt	7588	6632	11893
fill	5676	4720	8563

Tabelle 2: Größe des erzeugten Codes

Die Größe einiger RT-Beschreibungen ist in **Tabelle 2** aufgeführt. Bei den ersten beiden Beispielen `bsearch` und `bsort` handelt es sich um Implementierungen von Binärsuche und Bubblesort. Die folgenden drei Algorithmen `diffeq`, `gcd` und `ellipf` stammen aus der Sammlung HLSynth92 [1]. Von den letzten vier Beispielen führt `calc` einfache arithmetische Operationen in zwei aufeinanderfolgenden Schleifen aus, `max` bestimmt das Maximalelement innerhalb eines Arrays, `filt` enthält die inneren Schleifen eines FIR-Bildfilters und `fill` füllt einen Speicherbereich mit vorgegebenen Werten.
Während die unterschiedliche Behandlung der Signalverbindungen für C++ sich hinsichtlich des Implementierungsaufwandes kaum bemerkbar machte, ist dieser Effekt im generierten Code deutlich erkennbar. Dort wirkt er sich entsprechend der Anzahl der Signalverbindungen aus, was sich in der deutlich veränderten Codegröße im Vergleich zu VHDL und Verilog zeigt.

	VHDL	Verilog	C++
bsearch	2.29	0.89	100.68
bsort	2.42	0.93	121.18
diffeq	2.49	1.01	118.91
gcd	1.64	0.71	86.70
ellipf	2.29	0.81	96.42
calc	1.34	0.61	58.11
max	1.89	0.64	77.01
filt	2.34	0.83	102.81
fill	1.74	0.66	88.98

Tabelle 3: Kompilationszeiten

Für dieselben Beispiele wurden die Zeiten zur Kompilation der Schaltungsbeschreibungen gemessen (siehe **Tabelle 3**). Für VHDL verwendeten wir die Synopsys'97 Suite auf einer 300Mhz UltraSparc2. C++ wurde mit dem egcs-1.1.1 Compiler der Free Software Foundation auf derselben Maschine kompiliert. Für Verilog nutzten wir einen Simulator von Gateway Design Automation auf einer Sparc10. Trotz des langsameren Rechners erwies sich Verilog als am schnellsten. Die Kompilation von C++ hingegen benötigte extrem viel Zeit, was einerseits auf die Komplexität der Sprache zurückzuführen ist, andererseits aber auch auf die untypische Struktur des automatisch erzeugten Codes. Es muß berücksichtigt werden, daß diese Ergebnisse von den verwendeten Werkzeugen abhängen.

	VHDL	Verilog	C++
bsearch	6013	5553	6758
bsort	27320	22207	27452
diffeq	3940	3143	3523
gcd	3384	2751	3331
ellipf	1962	1620	2123
calc	1409	1202	1355
max	5051	4133	4819
filt	9241	7676	8906
fill	11161	8486	9068

Tabelle 4: Simulationsereignisse

In unserem C++ Simulator wird ein Prozeß ausgeführt, wenn eines der Ereignisse in der Prozeßaktivierungsliste des Prozesses eintritt. Bei mehreren Ereignissen wird der Prozeß entsprechend häufig ausgeführt.

Es werden keine Optimierungen zur Verringerung der Zahl der Prozeßaktivierungen bei der Abarbeitung der auszuführenden Prozesse vorgenommen.

Tabelle 4 zeigt die Zahl der Simulationsereignisse, die während eines Simulationslaufes bei jeder der Zielsprachen ausgegeben werden. Das unterschiedliche Verhalten der verschiedenen Simulatoren belegt die Notwendigkeit für eine Nachbearbeitung der Simulationsausgaben, um die Ergebnisse zwischen den unterschiedlichen Simulatoren vergleichbar zu machen.

	VHDL	Verilog	C++
bsearch	9.7	11.0	86.3
bsort	17.0	24.9	109.0
diffeq	7.5	8.0	72.7
gcd	10.8	10.8	102.8
ellipf	6.0	5.4	69.2
calc	9.1	8.0	116.9
max	15.1	16.4	132.8
filt	13.4	16.1	101.8
fill	24.0	27.2	183.6

Tabelle 5: Simulierte Taktzyklen/Sekunde

Schließlich zeigt **Tabelle 5** die Anzahl der simulierten Taktzyklen pro Sekunde. Trotz der langsamen Sparc10 zeigt Verilog recht gute Ergebnisse. Die Leistung des C++ Simulators erscheint vergleichsweise schlecht, da trotz Kompilation lediglich eine Steigerung um ungefähr den Faktor 10 verglichen mit dem auf der langsameren Architektur ausgeführten Verilog Simulator erreicht wurde. Eine Analyse ergab, daß zur Laufzeit viele zeitraubende Typumwandlungen stattfinden, was sich durch eine geschicktere Implementierung der Prozeß- und Signalklassen vermeiden ließe, unser Simulator also einiger Verbesserungen bedarf.

6 Fazit

Es wurde ein allgemeiner Ansatz zur Erzeugung von RT-Beschreibungen für unterschiedliche Zielsprachen vorgestellt. Pro Zielsprache ist nur ein kleiner Anteil an spezialisiertem Code notwendig, was den Wartungsaufwand des High-Level Synthesesystems gering hält.

Unsere Ergebnisse zeigen, daß die Unterstützung von Verilog als Ausgabesprache den geringsten Aufwand im Vergleich zu VHDL und C++ erforderte. Durch die Implementierung von C++ als weiterer Ausgabesprache wurde die Allgemeingültigkeit des Ansatzes belegt. Die Netzlistengenerierung in C++ erforderte jedoch die Bereitstellung einer zusätzlichen Laufzeitbibliothek und eines eigenen Simulators.

Das allen Zielsprachen gemeinsame Simulationsausgabeformat erlaubt eine sprachunabhängige Analyse der Simulationsergebnisse und bietet die Möglichkeit der sprachübergreifenden Validierung. Während unserer Simulationsläufe wurden auf diese Weise einige Fehler in der Implementierung der Funktionseinheiten aufgedeckt. Die Verwendung von Lisp-artigen Ein- und Ausgabeformaten ersparte uns einigen Entwicklungsaufwand, da auf spezielle Parser verzichtet werden konnte.

In unseren Experimenten erwies sich Verilog bei den verwendeten Werkzeugen als sehr effizient im Vergleich mit VHDL und C++. Bezüglich C++ zeigten die Ergebnisse, daß unser Simulator einiger Verbesserungen bedarf. Aber selbst wenn sich die C++ Simulationsleistung deutlich steigern ließe, machen die extrem langen Kompilationszeiten den Gebrauch von C++ erst für längere Simulationsläufe interessant.

Literatur

[1] HLSynth'92. http://www.cbl.ncsu.edu, 1992.

[2] A. W. Aho, R. Sethi, and J. D. Ullmann. *COMPILERS Principles, Techniques and Tools*. Addison Wesley, Massachusetts, 1986.

[3] B. Cohen. *VHDL Coding Styles and Methodologies*. Kluwer, 1995.

[4] Daniel D. Gajski, Nikil D. Dutt, A. C-H Wu, and S. Y-L Lin. *High-Level Synthesis — Introduction to Chip and System Design*. Kluwer, 1992.

[5] U. Holtmann. High–Level Synthese mit BSS für den Einsatz im Hardware/Software Codesign. In *6. E.I.S Workshop*, page 208ff, 1993.

[6] O. Levia, S. Maginot, and J. Rouillard. Lessons in language design: Cost/benefit analysis of VHDL features. In *DAC*, pages 447–453, 1994.

[7] G. De Micheli, D. Ku, F. Mailhot, and T. Truong. The olympus synthesis system. *IEEE Design & Test of Computers*, pages 37–53, October 1990.

[8] T.-T. Nguyen. THUD. http://www.glug.org/people/ttn/software/thud.

[9] D. J. Smith. VHDL & Verilog compared & contrasted — plus modeled example written in VHDL, Verilog and C. In *DAC*, pages 771–776, 1996.

[10] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, 2nd edition, 1993.

[11] D. E. Thomas and P. R. Moorby. *The Verilog Hardware Description Language*. Kluwer, June 1996.