Formale Modellextraktion von SystemC Entwürfen

Christian Genz, Universität Bremen, 28359 Bremen, Deutschland
Rolf Drechsler, Universität Bremen, 28359 Bremen, Deutschland
Gerhard Angst, Concept Engineering, 79111 Freiburg, Deutschland
Lothar Linhard, Concept Engineering, 79111 Freiburg, Deutschland

Kurzfassung

Die wachsende Beliebtheit der Systembeschreibungssprache SystemC trug während der letzten sieben Jahre kontinuierlich zur Erweiterungen der zugehörigen Bibliothek und zur Definition eines eigenen Standards durch IEEE bei. Allerdings nehmen sowohl die Flexibilität als auch die Mächtigkeit der Sprache Einfluss auf die Komplexität ihrer Analyse. So ist die Abbildung eines SystemC Modells auf ein statisches Modell, dessen Zustandsraum zur Laufzeit statisch ist, nicht immer möglich. In der vorliegenden Arbeit präsentieren wir ein Konzept zur Extraktion des Zustandsraumes zu einem gegebenen Zeitpunkt. Das vorgestellte Verfahren ist eine hybride Technik bestehend aus statischer und dynamischer Analyse. Da sich beide Techniken in diesem Ansatz ergänzen, bleibt die Hierarchie des Zustandsraumes erhalten. Eine Anwendung für diese Art der Analyse findet sich beispielsweise in Werkzeugen für Synthese und Visualisierung.

1 Einführung

Die Systembeschreibungssprache SystemC ist mittlerweile ein weitgehend anerkannter Standard für Entwürfe im Bereich des HW/SW Co-Designs. Anders als *Hardwarebeschreibungssprachen* (HDLs) bietet SystemC dem Systemarchitekten Konzepte und Techniken, welche zuvor nur in Software-Programmiersprachen zur Verfügung standen. Die breite Palette an Abstraktionsebenen, welche sowohl der Darstellung von HW-Strukturen als auch dem Spezifizieren von objektorientierten Algorithmen genügt, macht SystemC gleichermaßen für den industriellen und den wissenschaftlichen Einsatz wertvoll.
Die SystemC Entwurfsmethodik wirkt dem *design gap* entgegen, indem sie die Erstellung vom Prototypen eines Systementwurfes zum frühestmöglichen Zeitpunkt unterstützt. Somit kann die Entwicklungszeit signifikant reduziert werden, da mehrere Entwicklungsschritte des Systems - wie die Erstellung von Testumgebungen, die Synthese, die Implementierung von Anwendungen oder die Implementierung von Treibern - parallelisiert werden können.
Voraussetzung für diesen Geschwindigkeitszuwachs ist jedoch die Prämisse alle Vorteile von SystemC und der Sprache C++, auf der SystemC basiert, möglichst ohne Einschränkung benutzen zu können. Die Möglichkeit Algorithmen und Datenstrukturen beliebig wiederzuverwenden, sowie existierende Software-Bibliotheken einzubinden, muss in prototypischen HW/SW Co-Entwürfen gewährleistet sein.
Mit einer wachsenden Zahl von SystemC Nutzern stieg auch das Interesse an Werkzeugunterstützung. In Verbindung mit einigen auf dem Basispaket aufbauenden Bibliotheken wurde der Funktionsumfang von SystemC bereits erweitert um simulationsbasierte Verifikation [9], sowie eine Unterstützung zur Modellierung auf Transaktionsebene [2] und zur Modellierung analoger Schaltungen [15].
Einige nützliche Ansätze wie die Synthese [4], die Systemexploration [6], die Visualisierung [7] oder eine interaktive Debuggingunterstützung [14] binden zwangsläufig ein Analysekonzept ein, welches das Verhalten und die Struktur der Eingabespezifikation in eine für die jeweilige Anwendung günstige Repräsentation übersetzt. Ungünstigerweise bietet derzeit kein Analyseverfahren eine hinreichende Möglichkeit SystemC Entwürfe auf formale Modelle abzubilden, welche eine Unterstützung für die folgenden Konzepte bieten:

- Crossprobing
- HW/SW Co-Design
- hierarchische Darstellung des Zustandsraumes

Der hier vorgestellte Ansatz zur Analyse ist eine Weiterentwicklung der Verfahren aus [7] und [14] in dem die oben genannten Punkte voll unterstützt werden.
Die vorliegende Arbeit ist wie folgt gegliedert: in Abschnitt 2 wird sowohl ein Überblick als auch eine Abgrenzung zu existierenden Arbeiten gegeben. Anschließend werden in Abschnitt 3 eigene Vorarbeiten vorgestellt, welche die Basis dieser Arbeit bilden.
Da der in dieser Arbeit vorgestellte Ansatz einer Reihe von Transformationen beinhaltet, ist auch Abschnitt 4 in mehrere Teile untergliedert. Der erste dieser Unterabschnitte klärt die Frage nach der Notwendigkeit einer konkreten Zustandsextraktion. Im folgenden Unterabschnitt 4.2 wird auf die Architektur des Ansatzes einge-

gangen. In den verbleibenden Teilen des Abschnittes 4 werden die Details des Extraktionsverfahrens verdeutlicht. In Abschnitt 5 wird eine Fallstudie für die in Abschnitt 4 beschriebenen Extraktion gegeben. Die Arbeit wird durch eine Zusammenfassung in Abschnitt 6 beendet.

2 Verwandte Arbeiten

Um den Beitrag des hier vorgestellten Ansatzes deutlich darzustellen, wird zunächst ein Überblick existierender Herangehensweisen gegeben. Ein gemeinsames Merkmal aller hier genannten verwandten Ansätze ist die Verwendung der *SystemC data introspection*.
So verwenden Große et.al. in [8] den SystemC Kernel um Hierarchieinformation des Modells zum Simulationszeitpunkt zu erhalten. Jede Instanz eines Typs der von der Klasse `sc_object` abgeleitet wurde befindet sich nach der Evaluationsphase in einer vom SystemC Kernel verwalteten Liste. Durch das Traversieren aller Module und Signale der Liste und die entsprechenden Metainformationen bezüglich ihrer Konnektivität wird die Hierarchie des Modells nachvollzogen. Letztlich wird diese Hierarchie mittels einer GUI als Netzlistenstruktur abgebildet.
Der Ansatz von Eibl et. al. [3] baut auf einem sehr ähnlichen Prinzip wie [8] auf. Während Große et.al. jedoch auf einer direkten Modifikation der SystemC Bibliothek aufsetzt, basiert die Implementierung von [3] namens gSysC auf einer Wrapper-Bibliothek, welche die SystemC Bibliothek kapselt. Ebenso wie auch [8] ist gSysC nicht in der Lage über die Objektliste des Kernels hinaus Verhalten oder Strukturen zu erkennen, auch wenn diese wesentliche Teile des Entwurfes sind.
Im Gegensatz zu [8, 3] ist SystemCXML [1] ein Ansatz der nicht die *data introspection* zur Analyse verwendet. Die Extraktion der Hierarchie wird von Doxygen [16] durchgeführt. Nach dem Parsen des Quellcodes generiert Doxygen eine XML Repräsentation der entsprechenden Hierarchie. Von dieser wird wiederum eine Graphenbeschreibung im DOT Format erstellt. Dieser Graph kann mit dem Graphviz Paket angezeigt werden, welches frei verfügbar ist. Wie auch [8] und [3] ist SystemCXML nicht in der Lage hinreichend Verhalten zu extrahieren weshalb auch in diesem Ansatz die evaluierte Hierarchie unvollständig ist.
Ein weiteres Werkzeug welches jedoch zu reinen Analysezwecken entwickelt wurde ist KaSCpar [10] (Karlsruhe SystemC Parser), KaSCpar ist eine Sammlung mehrerer Programme die der syntaktischen Analyse, sowie einer anschließenden Elaboration dienen. KaSCpar leitet dabei einen AST von einem SystemC Modell ab. Dieser enthält nach der Analyse eine operationale Semantik des Programms. KaSCpar ist jedoch nicht in der Lage Zustandsräume aufzuzeichnen.
Wie auch KaSCpar ist PINAPA [12] ein Werkzeug zur reinen Analyse von SystemC Quellen. PINAPA baut auf der *GNU Compiler Collection* (GCC) auf. Anders als die zuvor präsentierten Arbeiten basiert PINAPA auf einer hybriden Form der Analyse. Das Ergebnis der statischen Analyse ist wie bei den anderen Ansätzen zuvor eine Hierarchie, welche jedoch auch das Verhalten beinhaltet. Die dynamische Analyse besteht aus einer SystemC Simulationsphase und einer anschließenden Abbildung der Hierarchieobjekte auf korrespondierende Speicheradressen. Dynamische Typen, welche auf Zeigern basieren und unvorhersehbar wachsen, bereiten PINAPA Probleme, da nur ihre Wurzel im Ergebnis der statischen Analyse sichtbar ist.
Unser Ansatz ist völlig *non-intrusiv*. Weder die zur Evaluation benötigte SystemC-Bibliothek, noch der verwendete Compiler werden modifiziert. Auch das Verhalten und die Architektur des analysierten SystemC Modells bleiben unverändert. Damit eignet sich unser Ansatz auch für dynamische Typen welche zur Laufzeit unvorhersehbar wachsen können.

3 Eigene Vorarbeiten

Beide Vorarbeiten [14, 7] basieren auf der selben interpretationsbasierten Technik, die zur Visualisierung der extrahierten Netzlistenstruktur genutz wurde. Der primäre Nachteil, der genannten Ansätze, spiegelt sich in der Verwendung der Abstraktionsebene wieder, welche die eingehende SystemC Schaltung charakterisiert. Während konkrete Entwurfsstile wie der *Register Transfer Level* (RTL) oder der *Gate Level* (GL) vollständig visuell dargestellt werden können, nimmt der Detailgrad der Darstellung mit einem zunehmenden Grad an Abstraktion ab.
Für die Visualisierung in beiden Ansätzen wird eine interaktive GUI namens RTLVision von Concept Engineering GmbH[1] verwendet. Diese stellt ein schematisches Bild der Schaltung wie in Abbildung 1 dar. Zudem erlaubt RTLVision die interaktive Exploration der Schaltung durch das Expandieren oder aber Kollabieren einzelner Module. Diese Technik kann jedoch nur

[1] www.concept.de

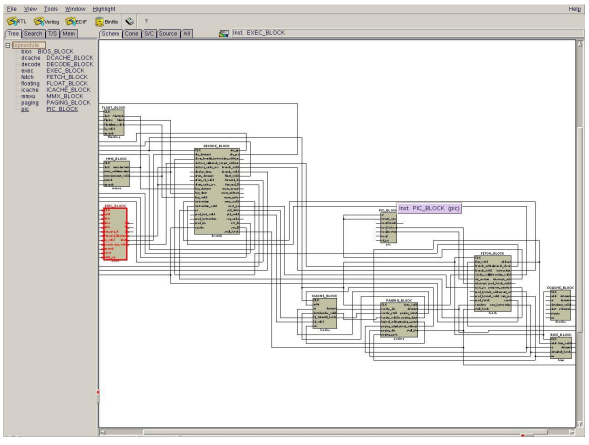

Bild 1 Darstellung einer Schaltung in RTLVision

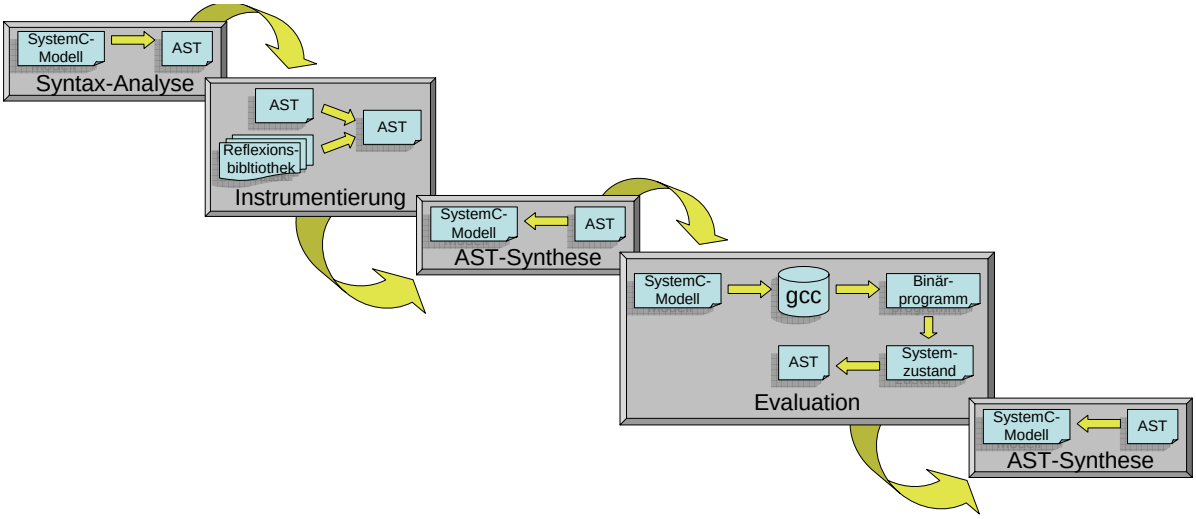

Bild 2 Architektur des Ansatzes

angewandt werden, wenn die in die GUI eingehende Schaltungsbeschreibung sowohl eine Struktur als auch ein Verhalten umfasst. Andernfalls werden leere Boxen dargestellt, da die Funktionalität an dieser Stelle unbekannt bleibt.

4 Zustandsextraktion

Zur Ableitung eines formalen Modells ist die Extraktion des Startzustandes notwendig, welcher Teil eines simulationsfähigen SystemC Modells ist. Der ermittelte Systemzustand beinhaltet konkrete Werte für alle Variablen über denen der Zustandsraum des Modells zum Startzeitpunkt der Simulation (`sc_start`) definiert ist.

4.1 Komplexität der Analyse

Dieser Unterabschnitt soll die Notwendigkeit der Zustandsextraktion motivieren. Dazu werden die bisher ungelösten Problemstellungen der interpretierenden Technik in den Vorarbeiten [7, 14] verdeutlicht. Im Anschluss folgt eine Beschreibung des Extraktionsverfahrens selbst.

1. Im Gegensatz zu HDLs wie VHDL und Verilog bietet SystemC explizite Unterstützung zur Modellierung des HW/SW Co-Designs. Basierend auf C++ gibt SystemC dem Architekten Zugriff auf Konzepte wie generische Typen oder die (De-)Allokation von Speicherbereichen zur Laufzeit.
   Infolgedessen ist die Größe des Zustandsraumes von SystemC Architekturen nicht zwingend konstant (anders als in HDL Beschreibungen). Eine direkte Abbildung oder Synthese eines solchen Modells mit einer sich verändernden Zahl an Zuständen und Zustandsübergängen auf einen *Deterministische Endliche Automaten* (DEA) bzw. eine entsprechende Schaltung ist nicht möglich (siehe [11, 5]). RTLVision, welches für die Visualisierung in [14, 7] verwendete wird, verlangt jedoch die Beschreibung einer Schaltung mittels der Verwendung von Symbolen der RT- oder der Gatter-Ebene.
2. Die Struktur einer SystemC Schaltung ergibt sich durch die Summe seiner Module und der verbundenen Signale oder auch Kanäle. Erst nach der Evaluationsphase, welche mit dem Aufruf der Funktion `sc_start` endet, ist diese Struktur dem Kernel bekannt.
   Die *SystemC data introspection* ist eine Technik welche den Zugriff auf die evaluierte Netzlistenstruktur eines Modells zur Laufzeit erlaubt und den Mangel an Reflexionsunterstützung von C++ ausgleichen soll. Das Debugging und die Visualisierung erfordern nebst den Wertigkeiten jedoch Informationen der Netzlistenelemente, welche durch den SystemC Kernel nicht gespeichert werden. Beispiele hierfür sind die Namenslisten aller SystemC Objekte, welche vom Kernel verwaltet werden und nicht zwangsweise mit den korrespondierenden Deklarationsnamen übereinstimmen müssen. Instanzen, die nicht Teil der SystemC Hierarchie sind, tauchen nicht in den Namenslisten auf.

4.2 Architektur

Das Extraktionsverfahren gliedert sich in vier Phasen, welche in Abbildung 2 dargestellt sind. Die syntaktische Analyse erlaubt die Abbildung eines SystemC Programms auf eine operationale Semantik in Form eines Syntaxbaumes. Der eigens für diese Applikation entwickelte Parser, sowie ein weiterer Scanner unterstützten spezielle Crossprobing Fähigkeiten und wurden mit Hilfe des *Purdue Compiler Construction Toolkit* (PCCTS) [13] entwickelt.

Der Syntaxbaum oder auch *Abstract Syntax Tree* (AST) dient zur Kommunikation unter den einzelnen Phasen. Da die Evaluation aber SystemC Spezifikationen als Eingabe verwendet, wurde die Umkehrfunktion des Parsers implementiert. Die AST-Synthese generiert aus einem Syntaxbaum ein SystemC Programm.
Damit das Ergebnis der Evaluation wieder in einem AST gespeichert wird, bedarf es einer Annotation des zu evaluierenden Modells. Diese Art der Annotation basiert auf automatisch generierten Funktionen. Die Generierung der entsprechenden Funktionen findet in der Instrumentierungsphase statt.
Ziel der Zustandsevaluation ist es generische Teile des AST auszuflachen und konkrete Wertebelegungen von Ausdrücken abzuleiten. Anders als die von SystemC gebotene *data introspection* expandiert diese Technik den zuvor vom Parser erstellten AST, anstatt Werteinformationen in den Datenstrukturen des SystemC Kernels zu speichern.

4.3 AST-Synthese

Der AST ist ein gerichteter Graph. Bestehend aus sechs verschiedenen Knotentypen bietet er die Möglichkeit beliebige Formen von Kontrollflussoperationen und Datenflussabhängigkeiten zu kombinieren. Auch Zustände lassen sich im AST darstellen. Durch die Speicherung von Deklarationen jeglichen Typs können Variablen, konstante Werte und Funktionen dargestellt werden.
Da die Semantik des Programms im AST gespeichert ist, kann diese mit der Manipulation des AST beliebig geändert oder auch erweitert werden. So müssen Wertebelegungen von Variablen lediglich durch einen Eintrag in der Struktur der Variable gesetzt oder geändert werden, was den Aufwand der Manipulationen gering hält.
Die Synthese des AST gestaltet sich weniger komplex als die syntaktische Analyse. Während die Analyse häufig mit Mehrdeutigkeiten bei der Ableitung des Syntaxbaumes konfrontiert wird, ist dieser selbst eindeutig.
Um die Crossprobing-Fähigkeiten des Visualisierungswerkzeuges von Concept Engineering nutzen zu können, speichert der AST neben Zeilennummern auch die Byteposition und den Dateinamen aller erkannten Tokens aus denen sich der AST zusammensetzt. Mittels dieser Informationen werden neue Dateien erzeugt, die der Eingabe der syntaktischen Analyse bis auf die Kommentare gleichen.

4.4 Instrumentierung

Die Instrumentierung des Quellcodes dient der Generierung von Funktionen, welche erst nach erneuter Übersetzung des Programms ausgeführt werden. Diese Funktionen werden im Folgenden Rekorderfunktionen genannt.
Jede Rekorderfunktion bewirkt die Speicherung einer Zustandsvariablen nach ihrer Wertänderung. Dazu expandiert die Rekorderfunktion den zur modifizierten Variable korrespondierenden Eintrag im AST zur Laufzeit. Da sich alle benutzerdefinierten Typen aus primitiven Typen (wie `int` und Zeigern) zusammensetzten, werden nur die Werteänderungen primitiv getypter Entitäten instrumentiert. Um Werteänderungen von Variablen ohne Seiteneffekte im AST zu speichern, wird die Propagation der Werteänderung direkt nach der evaluierten Werteänderung und noch vor der Evaluation benachbarter Ausdrücke durchgeführt.
Kommt es innerhalb des evaluierten Ausdruckes zum Wechsel des *stackframe* (z.B. Funktionsaufrufe, Blockanweisungen oder überladene Operatoren), so muss auch der dem *stackframe* entsprechende AST angepasst werden. Auch hierfür müssen Rekorderfunktionen generiert werden, die vor dem Betreten oder Verlassen eines Blocks den entsprechenden AST wachsen oder schrumpfen lassen.
Um zur Laufzeit des SystemC Simulators eine Kopie des AST expandieren zu können, wird eine Inklusionsdirektive für den Analyse-Quellcode in den AST generiert. Zusätzlich wird der AST um eine Instruktionssequenz erweitert. Die Sequenz veranlasst die statische (syntaktische) Analyse des zu evaluierenden Modells zum Ausführungszeitpunkt. Somit steht dem Simulator noch vor der Evaluationsphase eine exakte Kopie des AST zur Verfügung, welcher zur Instrumentierung verwendet wurde. Eine vereinfachte Darstellung der Kopplung von statischer und dynamischer Analyse durch die Instrumentierung kann Abbildung 3 entnommen werden.

4.5 Zustandsevaluation

Anstatt interpretierend zu evaluieren wie in [7] arbeitet dieser Ansatz nach der statischen Analyse simulativ. Die Simulation hat den Vorteil auch externe Bibliotheksfunktionen sowie Betriebssystemroutinen verwenden zu können. Die Extraktion von Werten ist jedoch nur möglich wenn diese im AST gespeichert wurden. Ein Beispiel, dass dies nicht zwangsweise der Fall ist, sind *arrays* und *void* Zeiger. Werden diese in Bibliotheksfunktionen beschrieben, so kann die Wertänderung nicht nachvollzogen werden, da der Typ des Wertes nicht bekannt ist. Selbst wenn nach der Zuweisung

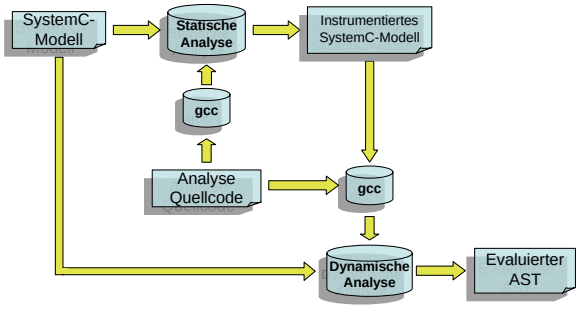

Bild 3 Hybride Analyse

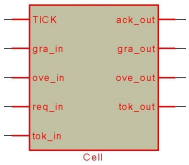

Bild 4 Zelle eines Arbiters

der Variable der Wert verfügbar ist, kann er nicht ohne Kenntnisse über seinen Typen interpretiert werden.
Der Extraktion voran steht die Evaluation. Diese wird jedoch durch den SystemC Kernel übernommen. Die evaluierten Werte der Zustandsvariablen werden durch die automatische Übersetzung des Instumentierungscodes ebenfalls automatisch in einem resultierenden AST gespeichert. Bei genauer Betrachtung ist der AST nach der Evaluationsphase ein Baum dessen Blätter konstanten Werten, Zustandsvariablen oder aber nicht definierten Funktionen entsprechen. Alle Daten- und Kontrollflussoperationen des AST befinden sich zwischen der Wurzel und den Blättern. Die Variablen, welche den Zustandsraum beschreiben, können folglich nach der Evaluation in *depth first search* Reihenfolge betrachtet und extrahiert werden.

5 Fallstudie

Um den auf den vorangegangenen Seiten erläuterten Ansatz in einer praktischen Anwendung zu demonstrieren, soll im Folgenden das Visualisierungswerkzeug aus [7] zur Veranschaulichung der extrahierten Hierarchie dienen. Sei das Modul `Cell` eine von n gleichartigen Zellen eines Arbiters. Jede Zelle des Arbiters besteht aus fünf Eingängen und vier Ausgängen. Jeder an den Arbiter angeschlossene *Client* kommuniziert mit dem Arbiter über die Schnittstelle des Moduls `Cell`. Die Zahl der vom Arbiter verwalteten Module lässt sich über den Parameter n definieren.

```
for (int i = 0; i < n; ++i )
{
  Cell *ncl = new Cell("cells");
  arbiter.cells->push_back( ncl );
}
```

Der Quellcode für die Instanziierung des Arbiter-Modells liest den Wert für n aus der Umgebung, die dem Programm zum Startzeitpunkt der Ausführung übergeben wurde. Anschließend wird n in einer Schleife während der Evaluationsphase zur Allokation neuer Zellen verwendet. Allein durch die statische Analyse ist nicht voraussagbar wie viele Zellen zur Laufzeit instanziiert werden. Das Ergebnis der Instanziierung kann in Abbildung 5 betrachtet werden.
Die Interna der Zellenimplementierung sind in Abbildung 6 wiedergegeben. Anders als in Abbildung 5 sind hier zwei Fensterbereiche zu erkennen. Während der obere Bereich ein schematisiertes Bild der Funktionalität einer Zelle und ihrer Gatter anzeigt, zeigt der untere Bereich den sogenannten *cone view*. Im *cone view* werden ausschließlich selektierte Elemente und Pfade angezeigt.

6 Zusammenfassung

In der vorliegenden Arbeit präsentierten wir eine Analysetechnik, die die Modellextraktion von SystemC Modellen auch auf hohem Abstraktionsniveau ermöglicht. Dabei wurden Details bezüglich der Struktur und des Verhaltens der entsprechenden Schaltung nicht vernachlässigt. Es wurde auf die Innovationen gegenüber den Vorarbeiten [14, 7] eingegangen, sowie die Notwendigkeit des Analyseschrittes, um diese zu erreichen. Zukünftige Arbeiten auf diesem Gebiet werden die bisher entwickelte Analysetechnik erweitern, um sie zu Zwecken des Profilings von Busarchitekturen zu nutzen. Die Resultate dieser Arbeit werden sich speziell auf die Visualisierung und Synthese von Kommunikationsarchitekturen konzentrieren.

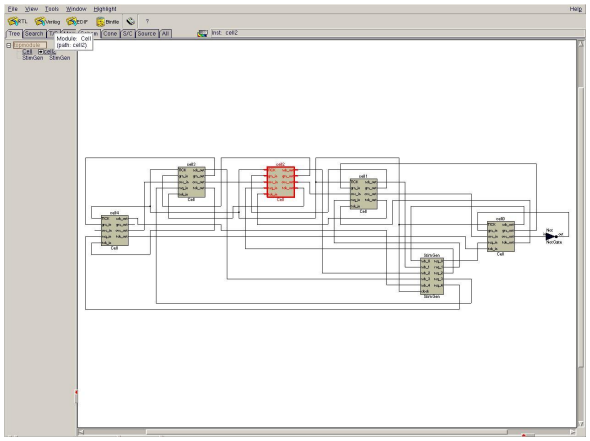

Bild 5 Instanziierung des Arbiters mit fünf Zellen

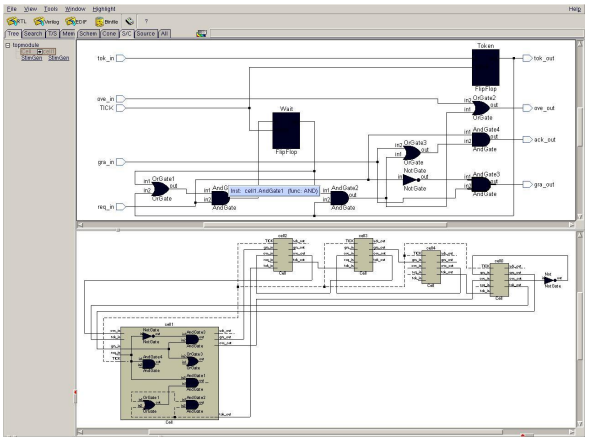

Bild 6 Implementierung einer Zelle

7 Literatur

[1] D. Berner, H. Patel, D. Mathaikutty, J.-P. Talpin, and S. Shukla. SystemCXML: An extensible SystemC front end using XML. Technical Report 06, Virginia Polytechnic Institute and State University, 2005.

[2] L. Cai and D. Gajski. Transaction level modeling: an overview. In *IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS)*, pages 19–24, 2003.

[3] C. Eibl, C. Albrecht, and R. Hagenau. gSysC: A graphical front end for SystemC. In *European Conference on Modelling and Simulation*, pages 257–262, 2005.

[4] G. Fey, D. Große, T. Cassens, C. Genz, T. Warode, and R. Drechsler. ParSyC: An efficient SystemC parser. In *Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI)*, pages 148–154, 2004.

[5] D. D. Gajksi, N. Dutt, S. Y.-L. Lin, and A. Wu. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[6] C. Genz and R. Drechsler. System exploration of SystemC designs. In *IEEE Annual Symposium on VLSI*, pages 335–340, 2006.

[7] C. Genz, R. Drechsler, G. Angst, and L. Linhard. Visualization of SystemC Designs. In *IEEE International Symposium on Circuits and Systems*, pages 413–416, 2007.

[8] D. Große, R. Drechsler, L. Linhard, and G. Angst. Efficient automatic visualization of SystemC designs. In *Forum on Specification and Design Languages*, pages 646–657, 2003.

[9] C. Ip and S. Swan. A tutorial introduction on the new SystemC verification standard, 2003. Available at http://www.systemc.org.

[10] F. Karlsruhe. KaSCpar. http://www.fzi.de/sim/kascpar.html.

[11] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.

[12] M. Moy, F. Maraninchi, and L. Maillet-Contoz. PINAPA: An extraction tool for SystemC descriptions of systems-on-a-chip. In *ACM international conference on Embedded software*, pages 317–324, 2005.

[13] T. Parr. *Language Translation using PCCTS and C++: A Reference Guide*. Automata Publishing Company, 1997.

[14] F. Rogin, C. Genz, R. Drechsler, and S. Rühlke. An Integrated SystemC Debugging Environment. In *Forum on Specification and Design Languages*, 2007.

[15] A. Vachoux, C. Grimm, and K. Einwich. SystemC-AMS requirements, design objectives and rationale. In *Design, Automation and Test in Europe*, pages 388–393, 2003.

[16] D. van Heesch. Doxygen. http://de.wikipedia.org/wiki/Doxygen.