Formale Verifikation eingebetteter Systeme

Detlef Schmid, Klaus Schneider, Michaela Huhn, George Logothetis und Viktor Sabelfeld
Institut für Rechnerentwurf und Fehlertoleranz (IRF), Universität Karlsruhe

von links nach rechts:

Dr. rer. nat. Viktor Sabelfeld studierte von 1966 bis 1971 Mathematik und arbeitete dann bis 1996 an der Universität Nowosibirsk als Professor. Sein Forschungsgebiet liegt auf den äquivalenten Programmtransformationen. Seit November 1997 arbeitet er am IRF.

Dr. rer. nat. Michaela Huhn studierte 1985-1992 Informatik in Erlangen und arbeitete bis 1997 an der Universität Hildesheim im Themengebiet "'Spezifikation und Verifikation verteilter Systeme"'. Sie promovierte über die Integration von Verfeinerung in prozeßalgebraischen und temporallogischen Spezifikationen. Seit 1998 arbeitet sie am IRF.

Prof. Dr.-Ing. Detlef Schmid ist Mitbegründer der Fakultät Informatik der Universität Karlsruhe und leitet seitdem das Institut für Rechnerentwurf und Fehlertoleranz (IRF). Mit dem Sonderforschungsbereich "'Automatisierter Systementwurf"' begann er bereits 1992, sich intensiv mit formaler Verifikation zu beschäftigen.

Dr. rer. nat. Klaus Schneider studierte von 1987-1992 Informatik an der Universität Karlsruhe und promovierte 1995 über Abstraktionsverfahren in der Hardwareverifikation bei Prof. Dr.-Ing. Schmid. Seit 1997 koordiniert er das Projekt "'Verifikation eingebetteter Systeme"' des Schwerpunktprogramms.

Dipl.-Inform. George Logothetis studierte Informatik an der Universität Frankfurt. In seiner Diplomarbeit entwickelte er Verfahren zur Modellprüfung von temporalen Logiken mit quantitativer Zeit. Seit November 1997 arbeitet er am IRF.

Wir beschreiben das an unserem Institut entwickelte Verifikationswerkzeug C@S*, welches speziell für die formale Verifikation reaktiver eingebetteter Systeme entwickelt wurde.* C@S *basiert auf einer neu entwickelten synchronen Sprache PURR, deren Konzepte detailliert erläutert werden.*

Formal Verification of Embedded Systems

We describe the verification system C@S *developed at our institute that has been designed for the verification of reactive embedded systems.* C@S *is based on a new synchronous language PURR, whose concepts are discussed in detail.*

1 Einführung

Heute spielt der Entwurf von eingebetteten Systemen, also von informationsverarbeitenden Komponenten, die in ein umgebendes System (etwa ein Auto) integriert sind, eine immer größere Rolle. Eingebettete Systeme sind oft heterogen aufgebaut, d.h. sie bestehen sowohl aus Software als auch aus Hardware.

Während im Softwareentwurf mit Hilfe von Rapid Prototyping oft frühzeitig die Funktionalität des zu entwerfenden Systems validiert werden kann, sind die Möglichkeiten beim Hardwareentwurf diesbezüglich eingeschränkt. Daher wird im digitalen Hardwareentwurf das Verhalten der Schaltungen vor der Fertigung mit Hilfe von umfangreichen Simulationsläufen untersucht. Eine vollständige Simulation der Hard- und Softwarekomponenten eines eingebetteten Systems ist aber aufgrund der Komplexität in der Regel zu aufwendig. Aus diesem Grund spielt die formale Verifikation bei eingebetteten Systemen eine noch größere Rolle als beim reinen Hard- oder Softwareentwurf.

Unter *formaler Verifikation* versteht man den mathematischen Nachweis, daß ein System gewisse Anforderungen erfüllt. Setzt man Verifikation in frühen Entwurfsphasen ein, sind Fehler vor Beginn der Produktion erkennbar und somit kostengünstig behebbar. Damit können Verifikationsmethoden einen entscheidenden Beitrag zur Senkung der Entwurfskosten leisten [1, 2], obwohl der Einsatz der Verifikation zunächst einen zusätzlichen Entwurfsschritt darstellt.

Grundlage der formalen Verifikation ist eine mathematische Beschreibung des Systems und der nachzuweisenden Eigenschaften. Die Auswahl der Beschreibungsmittel wird durch die Charakteristika eingebetteter Systeme bestimmt: Aus informatischer Sicht gehören eingebettete Systeme zu den sogenannten *reaktiven* Systemen. Ein reaktives System reagiert kontinuierlich mit Reaktionen (Ausgaben) auf Aktionen (Eingaben) der Umgebung [3, 4]. Die Zeitpunkte, zu denen die Eingaben erfolgen, werden ausschließlich von der Umgebung bestimmt. Reaktive Systeme unterscheiden sich somit von *interaktiven Systemen*, die zwar auch kontinuierlich mit ihrer Umgebung kommunizieren, aber die Interaktionszeitpunkte selbst bestimmen.

Reaktive Systeme unterliegen somit gewissen Echtzeitanforderungen, da jede Reaktion abgeschlossen sein muß, bevor die nächste Aktion der Umgebung erfolgt. Um diese Echtzeitbedingungen erfüllen zu können, besteht die Software in der Regel aus mehreren Prozessen. Auf der Hardwareseite werden neben mehreren Prozessoren oft auch zusätzliche Spezialkomponenten verwendet.

Neuere Untersuchungen zeigen, daß *synchrone* Sprachen für die formale Beschreibung und die Verifikation eingebetteter Systeme gut geeignet sind. Die Grundlagen synchroner Sprachen werden im nächsten Abschnitt diskutiert. In Abschnitt 3 werden die Konzepte der von uns entwickelten neuen synchronen Modellierungssprache PURR erläutert. Das auf dieser Sprache basierende Verifikationssystem C@S wird dann im letzten Abschnitt skizziert. Weitere Informationen sind im WWW unter der URL http://goethe.ira.uka.de/hvg/cats/ zu finden.

2 Synchrone Sprachen

Zur Beschreibung von eingebetteten Systemen werden in zunehmendem Maße synchrone Sprachen [4] verwendet. Diese Sprachen erlauben, Parallelität auf der Ebene leichtgewichtiger Prozesse zu beschreiben. Die Anweisungen synchroner Programmier- und Modellierungssprachen unterteilen sich in zeitverbrauchende und zeitlose. Jede zeitverbrauchende Anweisung benötigt ein Vielfaches einer fest gewählten logischen "‘Zeiteinheit"’, so daß alle Prozesse synchron zueinander arbeiten.

Die Kommunikation zwischen den Prozessen erfolgt in der Regel durch Senden von global sichtbaren Signalen, die auch Daten tragen können. Dieses elementare Kommunikationsprinzip erlaubt auch die Implementierung anderer Kommunikationsarten wie etwa Datenaustausch über Kanäle oder gemeinsame Daten.

Graphische synchrone Sprachen wie z.B. Statecharts [5], Argos [6] und SyncCharts [7] entstanden früh aus der Modellierung von endlichen Automaten. Im Gegensatz zu konventionellen endlichen Automaten können hier nebenläufige kommunizierende Automaten beschrieben werden, deren Zustände selbst wieder Automaten enthalten können. Insbesondere Statecharts haben für den Entwurf eingebetteter Systeme bereits eine weite Verbreitung in der Industrie gefunden.

Ferner wurden auch aus dem Bereich der Datenflußsprachen synchrone Sprachen entwickelt. Die derzeit wichtigsten Vertreter sind Lustre [8] und SIGNAL [9]. Diese Sprachen fassen Variablen und Terme als Datenströme auf. Programme sind prinzipiell Gleichungssysteme, die die Datenströme rekursiv in Abhängigkeit von den momentanen sowie den unmittelbar vorangegangenen Werten bestimmen. Diese Sprachen verwenden Konzepte aus der Regelungstechnik, speziell aus dem Bereich der rückgekoppelten Systeme.

Einen dritten Zweig bilden imperative synchrone Sprachen wie z.B. ESTEREL [10, 11], Reactive-C (RC) [12] (eine synchrone Erweiterung der Sprache C) und SML [13] (synchronous modelling language). Bei imperativen synchronen Sprachen besteht eine Reaktion in der Ausführung eines Codesegments zwischen zwei "‘Kontrollpunkten"’, d.h. Stellen im Programmtext, bei denen eine Zeiteinheit verbraucht werden soll, bzw. Stellen, an denen sich verschiedene Kontrollflüsse synchronisieren. Jedem Prozeß ist zu einem Zeitpunkt genau ein Kontrollpunkt zugeordnet.

Speziell ESTEREL bietet für den Entwurf eingebetteter Systeme wichtige Konstrukte zur Unterbrechungsbehandlung. Diese Konstrukte gewährleisten, daß eine Unterbrechung sofort, d.h. noch während der momentanen Interaktion stattfindet. Dieses Verhalten kann ansonsten nur mit Hilfe von Echtzeitbetriebssystemen garantiert werden, was bei ESTEREL jedoch nicht notwendig ist.

Ferner wurden für ESTEREL spezielle Algorithmen zur Software- und Hardwaresynthese entwickelt. Bei der Softwaresynthese werden ESTEREL-Module, die aus sehr vielen Prozessen bestehen können, auf einen einzigen sequentiellen Prozeß abgebildet. Der dabei entstehende Prozeß kann dann als C-Programm ausgegeben werden und steht somit für nahezu alle Mikrokontroller zur Verfügung. Besonders vorteilhaft ist, daß dabei kein (Echtzeit)-Betriebssystemkern zur Prozeßverwaltung mehr notwendig ist, der einen hohen Anteil der für die Software verfügbaren Ressourcen beanspruchen würde.

Daneben wurden auch Methoden zur direkten Hard-

waresynthese aus ESTEREL-Programmen entwickelt, um die vorhandene Parallelität direkt nutzen zu können. Häufig sind die so entwickelten Schaltungen auch ohne weitere Optimierungen sehr effizient [14].

Die zur Synthese von ESTEREL-Programmen verwendeten Techniken sind als korrekt nachgewiesen worden, so daß sie "'formale Syntheseschritte"' darstellen, deren Resultate per Konstruktion zum gegebenen ESTEREL-Entwurf äquivalent sind. Somit kann sich die Verifikation vollständig auf ESTEREL-Entwürfe beschränken.

Für die Verifikation bieten synchrone Sprachen entscheidende Vorteile: während für reale Zeitmodelle aufgrund von Unentscheidbarkeitsresultaten kaum Rechnerunterstützung zu erwarten ist, eignen sich Systembeschreibungen auf synchronen, logischen Zeitmodellen gut für eine automatische Verifikation.

Logische Zeitmodelle stehen in engem Zusammenhang mit dem realen Zeitverbrauch, so daß Rückschlüsse auf das reale Echtzeitverhalten möglich sind: Bei der Hardwaresynthese können heute verfügbare Werkzeuge die minimalen Taktzeiten ermitteln und ggf. Optimierungen vornehmen. Nach der Softwaresynthese von ESTEREL kann die maximale Anzahl an Maschinenbefehlen bestimmt werden, die während einer Interaktion auszuführen ist, da innerhalb einer Interaktion keine datenabhängige Schleife vorkommen kann. Zusammen mit den Kenndaten des Mikroprozessors ergeben sich somit in einfacher Weise Abschätzungen für reale Zeitschranken.

3 PURR

Die Zielvorstellung, möglichst viele Aspekte des Systemverhaltens verifizieren zu können, erfordert trotz der erwähnten Vorteile Erweiterungen in den existierenden synchronen Sprachen, die in unserem Projekt zur Entwicklung einer eigenen Sprache PURR geführt haben.

Benutzerdefinierte Datentypen sind für Systembeschreibungen auf höheren Abstraktionsebenen sehr nützlich. Leider können benutzerdefinierte Datentypen in ESTEREL nicht definiert werden. Statt dessen werden sie aus einer anderen imperativen Sprache importiert. Für den Systementwurf ist dieses Konzept ausreichend, für die Verifikation muß bei diesem Ansatz allerdings neben der ESTEREL-Semantik auch die Semantik dieser sogenannten "'host language"' formalisiert werden.

Daher wurden als erste Ergänzung von ESTEREL zu PURR Konstrukte zur Definition abstrakter Datentypen und darauf arbeitenden Funktionen aufgenommen. Die zur Verfügung gestellten Konstrukte sind so aufgebaut, daß Verifikationsziele, die sich auf diese Datentypen beziehen, gut mit klassischen Methoden der Termersetzung [15] bewältigt werden können. Eine genauere Darstellung der Datentypdefinitionen in PURR findet man in [16].

Nichtdeterminismus: Bei der Implementierung eingebetteter Systeme soll jede Komponente in Abhängigkeit von den Eingaben und vom internen Zustand deterministisch reagieren, andernfalls wäre das Verhalten komplexer Systeme kaum durchschaubar. Im Gegensatz dazu tritt Nichtdeterminismus bei der Modellierung auf, sobald von bestimmten Implementationsdetails abstrahiert wird. Insbesondere hierarchische Verifikationsverfahren, die zur Verminderung der Problemkomplexität bei großen Systemen eingesetzt werden müssen, erzeugen nichtdeterministische Systeme.

Da ESTEREL auf die Beschreibung deterministischer Systeme beschränkt ist, wurde in PURR eine Erweiterung um Nichtdeterminismus vorgenommen: Mit dem Auswahlkonstrukt $\varepsilon x.M(x)$ kann ein Wert nichtdeterministisch aus einer Menge $M(x)$ ausgewählt werden. Beispielsweise kann man mit $\varepsilon x.x \bmod 3 = 0$ eine durch 3 teilbare Zahl auswählen. Da die nachfolgenden Ausführungsschritte von einer solchen Auswahl abhängen können, läßt sich so beliebiges nichtdeterministisches Verhalten modellieren.

Darüber hinaus erlaubt PURR auch Nichtdeterminismus im zeitlichen Verhalten, indem über den Auswahloperator nichtdeterministisch die Anzahl der zu verbrauchenden Zeiteinheiten bestimmt wird. Beispielsweise benötigt die PURR-Anweisung `pause [3,5]` $\equiv$ `pause` $\varepsilon x.3 \leq x \wedge x \leq 5$ entweder 3, 4, oder 5 Zeiteinheiten.

Automaten und Strukturen: Für Hardwarekomponenten liegen oft schon optimierte Komponenten vor, die direkt übernommen werden sollen. Daher bietet PURR die Möglichkeit, Hardwareentwürfe auf Register-Transfer-Ebene zu beschreiben. Basismodule können in PURR direkt als endliche Automaten dargestellt werden. Komplexere Schaltungen, die sich aus der Verschaltung bereits vorhandener Komponenten ergeben, lassen sich durch spezielle Strukturmodule in PURR definieren.

Mit Hilfe der Strukturmodule können auch *generische* Schaltungen erzeugt werden, die einen regulären Aufbau haben, der von einem Parameter des Moduls gesteuert wird. Beispiele für generische Strukturen sind arithmetische Schaltungen, die regulär bzgl. der Bitbreite aufgebaut sind, systolische Felder, die oft in eingebetteten Systemen zur Signalverarbeitung verwendet werden, aber auch Softwarekomponenten wie Protokolle, die die Kommunikation einer Anzahl von Teilnehmern steuern.

Spezifikationen: Neben der formalen Beschreibung

des Systems müssen auch die nachzuweisenden Eigenschaften formalisiert werden. PURR stellt dazu eine Logik höherer Stufe zur Verfügung, die um spezielle Konstrukte wie etwa temporale Operatoren erweitert wurde. Da diese Spezifikationssprache eine Obermenge der meisten gebräuchlichen Formalismen ist, können einerseits unterschiedliche Anforderungen komfortabel ausgedrückt werden, andererseits ist eine formal saubere Kombination der Verifikationsmethoden, die hinter den verschiedenen Formalismen stehen, möglich.

Die in Logik höherer Stufe spezifizierten Eigenschaften werden in PURR in drei Kategorien unterteilt:

Umgebungsannahmen beeinflussen die Abarbeitung der Anweisungen des Moduls selbst nicht, aber die Verifikation garantiert die Einhaltung der Spezifikationen nur dann, wenn diese Umgebungsannahmen erfüllt sind.

Verhaltensrestriktionen schränken das gemäß der Systembeschreibung mögliche Verhalten durch zusätzliche Anforderungen ein. So kann eine Fairneßrestriktion eine nichtdeterministische Auswahl so einschränken, daß nicht immer der gleiche Wert ausgewählt wird.

Spezifikationen beschreiben die geforderten Eigenschaften, die von der Verifikation unter den gegebenen Umgebungsannahmen und Verhaltensrestriktionen nachzuweisen sind.

Die genannten Erweiterungen, d.h. benutzerdefinierte Datentypen, Nichtdeterminismus, Schaltungsstrukturen mit Konstrukten zur Beschreibung regulärer Module und Logik höherer Stufe als Spezifikationssprache zeichnen PURR im Gegensatz zu ESTEREL eher als Modellierungssprache aus, die in bezug auf die Verifikation optimiert wurde.

4 Das C@S Verifikationssystem

Die heute als ausgereift geltenden Verifikationsverfahren lassen sich grob in *Modellprüfungsverfahren* und in *Theorembeweisen* unterteilen. Während erstere bevorzugt für die Verifikation des Kontrollflusses bei reaktiven Systemen eingesetzt werden, sind letztere für die Verifikation des Datenflusses besser geeignet. Ein Überblick über die gebräuchlichsten Formalismen geben [17, 18] und vor allem [19].

Die *Verifikation des Kontrollflusses* führt in der Regel auf entscheidbare Probleme, die sich auf die Korrektheit der parallelen Abläufe beziehen. Daher liegt die Verwendung von temporalen Logiken [20], welche Aussagen über zeitliche Relationen von Ereignissen beschreiben können, nahe. Die Modellprüfung temporaler Logiken dominiert im Moment diesen Bereich der Verifikation: Mit effizienten Werkzeugen wie SMV[21] wurden bereits viele Systeme wie DMA-Steuerungen [21], Ampelsteuerungen [22], Warteschlangen [22] sowie Protokolle [21, 23] verifiziert. Dabei wurden zum Teil auch Fehler in den Entwürfen entdeckt.

Die *Verifikation des Datenflusses* führt hingegen in der Regel auf unentscheidbare Probleme und erfordert eine grundsätzlich andere Vorgehensweise. Datenpfade werden meist mit Prädikatenlogik erster oder höherer Stufe beschrieben und durch den Einsatz von interaktiven Theorembeweisern, wie z.B. Nqthm [24], RRL [25], HOL [26] oder PVS [27] verifiziert. Diese Theorembeweiser wurden von verschiedenen Forschungsgruppen bereits eingesetzt (etwa zur Verifikation von Prozessoren [28, 29, 30, 31]). Auch wenn in diesem Bereich keine vollständige Automatisierung erzielbar ist, so läßt sich durch geeignete Strukturierung der Beweisziele eine wenigstens teilweise Automatisierung erreichen [32].

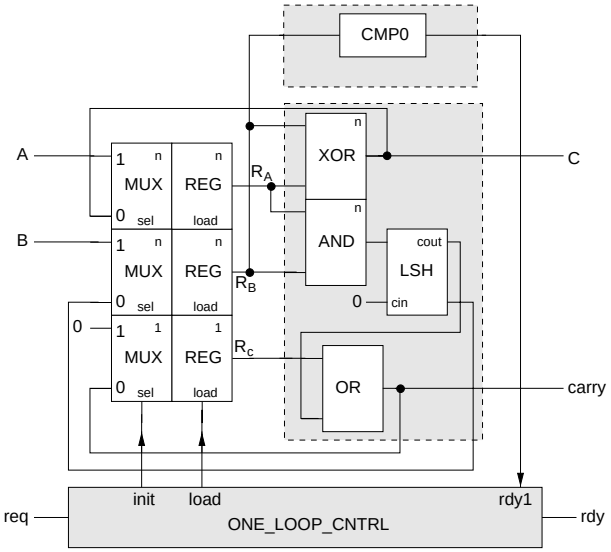
Bei eingebetteten Systemen treten allerdings häufig Verifikationsaufgaben aus beiden Problemklassen auf. Aus diesem Grund wurde unser Verifikationswerkzeug C@S nicht als monolithisches System für eine bestimmte Verifikationsmethode konzipiert, sondern erlaubt eine flexible Integration bereits vorhandener Verifikationswerkzeuge. So wurden in C@S neben dem Modellprüfer SMV das Termersetzungssystem RRL sowie die NP-Tools [33] zum aussagenlogischen Tautologietest integriert. Neben temporalen Logiken können in C@S auch direkt endliche Automaten zur Spezifikation des Kontrollflusses verwendet werden, so daß auch eine Integration von graphischen Sprachen wie Statecharts in Zukunft denkbar ist. Eingeflossen sind auch eigene Entwicklungen zu Varianten von Temporallogiken [34] und speziellen Modellprüfungsverfahren [35, 36].

Die Vielzahl der in C@S verfügbaren Verifikationsmethoden erlaubt dem Benutzer, das dem Problem am besten geeignete Verfahren auszuwählen und so mit möglichst geringem Aufwand zu Ergebnissen zu gelangen.

C@S bietet nicht nur eine Schnittstelle zu den genannten Verifikationswerkzeugen, sondern unterstützt auch eine *kompositionelle* Vorgehensweise: Komplexe Verifikationsziele können durch vordefinierte Regelanwendungen vom Benutzer manipuliert und zerlegt werden. Für die Unterziele können dann verschiedene Verfahren eingesetzt werden.

Die Verifikation in der C@S-Umgebung soll kurz an folgendem Beispiel einer regulären Hardwarestruk-

tur erläutert werden.



Bei obiger Schaltung handelt es sich um einen sequentiellen Addierer, dessen Prinzip auf von Neumann zurückgeht. Ist die Schaltung bereit (Ausgang rdy), so werden bei req Zahlen A und B gelesen. Die Addition dieser Zahlen erfolgt durch eine Iteration, deren Terminierung durch rdy angezeigt wird. Während der Iteration können keine weiteren Berechnungen angefordert werden ($rdy = 0$).

Bei der Verifikation wird man zunächst versuchen, auf eine automatische Methode zurückzugreifen. Im vorliegenden Fall konnten wir mit Modellprüfungsalgorithmen die Korrektheit bis zu einer Bitbreite von etwa 35 nachweisen, ohne daß größere Benutzerinteraktionen notwendig wurden. Die Rechenzeiten bewegten sich im Rahmen von 1-2 Minuten. Da man für feste Bitbreiten die maximale Rechenzeit dieser Schaltung kennt, ist es möglich, den Schleifenrumpf durch eine spezielle Transformation von C@S "'auszurollen"'. Die Modellprüfung wird dadurch effizienter, so daß die Verifikation der 64-Bit breiten Schaltung möglich wurde.

Alternativ kann der Benutzer die Verifikation auch durch weitere Kenntnisse über den Entwurf unterstützen. Bei obigem Beispiel kann man etwa als Invariante angeben, daß die Summe der Register während der Iteration konstant bleibt. Spezielle Beweisregeln in C@S erlauben dann eine Reduktion des ursprünglichen Verifikationszieles, so daß die Betrachtung eines einzigen Schleifendurchlaufs ausreicht. Durch diese Benutzerinteraktion kann die Schaltung mit Modellprüfung für bis zu 128 Bits als korrekt bewiesen werden. Die reine Rechenzeit beträgt dann etwa 1 Minute.

Durch die Integration von Induktionsbeweisern wie RRL ist in C@S auch die Verifikation für alle Bitbreiten per Induktion möglich. Da es sich dabei um ein stark interaktives Verfahren handelt, sind hier Rechenzeiten uninteressant. Zur Verifikation der obigen Schaltung benötigt ein eingearbeiteter Benutzer etwa einen Tag. Für das hier vorgestellte Beispiel ist diese Methode daher nicht besonders vorteilhaft. Dennoch existieren viele Beispiele, die mit Modellprüfung nur unzureichend behandelt werden können. In diesen Fällen erlaubt C@S dennoch eine, wenn auch interaktive, Verifikation mittels Theorembeweisern.

Obwohl C@S in erster Linie ein Verifikationssystem darstellt, ist für die Zukunft auch eine Eingliederung in herkömmliche Entwurfsabläufe denkbar. So bietet es sich an, Umgebungsannahmen und Verhaltensrestriktionen stärker im Entwurf zu berücksichtigen, um die Hardware/Software-Synthese zu optimieren.

Danksagung

Wir danken unseren Kollegen Dr. Th. Kropf und J. Ruf für ihre Unterstützung unserer Arbeit.

References

[1] A. Hall. Seven myths of formal methods. *IEEE Software*, pp. 11–19, 1990.

[2] J.P. Bowen und M.G. Hinchey. Seven more myths of formal methods. Technical Report PRG-TR-7-94, Oxford University Computing Laboratory, 1994.

[3] D. Harel und A. Pnueli. *Logic and Models of Concurrent Systems*, chapter On the development of reactive systems. NATO Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems. Springer Verlag, 1985.

[4] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Publishers, 1993.

[5] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computing*, pp. 231–274, 1987.

[6] F. Maraninchi. Argonaute: graphical description, semantics and verification of reactive systems by using a process algebra. *International Workshop on Automatic Verification Methods for Finite State Systems*. Springer Verlag, LNCS 407, 1989.

[7] Ch. Andre. Syncharts: A visual representation of reactive behaviors. Rapport de recherche tr95-52, Université de Nice-Sophia Antipolis, 1995.

[8] N. Halbwachs, P. Caspi, P. Raymond und D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[9] T. Gauthier, P. Le Guernic und L. Besnard. SIGNAL, a declarative language for synchronous programming of real-time systems. *Conference on Functional Programming Languages and Computer Architecture*. Springer Verlag, LNCS 274, 1987.

[10] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[11] G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling und M. Tofte, Editoren, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.

[12] F. Boussinot. Reactive C: An extension of C to program reactive systems. *Software Practice and Experience*, 21(4):401–428, 1991.

[13] M.C. Browne and E.M. Clarke. SML: A high level language for the design and verification of finite state machines. *IFIP WG 10.2 International Working Conference From HDL Descriptions to Guaranteed Correct Circuit Designs*, Grenoble, France, 1986.

[14] H. Touati and G. Berry. Optimized controller synthesis using Esterel. *International Workshop on Logic Synthesis*, Lake Tahoe, 1993. IEEE Computer Society Press.

[15] D. Hofbauer and R.-D. Kutsche. Proving inductive theorems based on term rewriting systems. *International Workshop on Algebraic and Logic Programming*, pp. 180–190, 1988.

[16] T. Kropf, J. Ruf, K. Schneider und M. Wild. A synchronous language for modeling and verifying real time and embedded systems. *GI/ITG/GME Workshop: Methoden des Entwurfs und der Verifikation digitaler Schaltungen und Systeme und Beschreibungssprachen und Modellierung von Schaltungen und Systemen*. HNI-Verlagsschriften, ISBN 3-931466-35-3, 1998.

[17] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In J.W. Baker, W.-P. de Roever und G. Rozenberg, Editoren, *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*, pp. 510–584, New-York, 1986. Springer Verlag.

[18] M. Yoeli. Formal verification of hardware design. Technical report, IEEE Computer Society Press, Los Alamitos, 1990.

[19] A. Gupta. Formal hardware verification methods: A survey. *Journal of Formal Methods in System Design*, 1:151–238, 1992.

[20] E.A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, Editor, *Handbook of Theoretical Computer Science*, volume B, pp. 996–1072, Amsterdam, 1990. Elsevier Science Publishers.

[21] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[22] M.C. Browne, E.M. Clarke, D.L. Dill und B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers*, C-35(12):1034–1044, 1986.

[23] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan und L.A. Ness. Verification of the Futurebus+ Cache Coherence Protocol. In D. Agnew, L. Claesen und R. Camposano, Editoren, *IFIP Conference on Computer Hardware Description Languages and their Applications (CHDL)*, pp. 5–20, Ottawa, Canada, 1993. Elsevier Science Publishers B.V., Amsterdam, Netherland.

[24] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[25] D. Kapur and H. Zhang. RRL: a rewrite rule laboratory. In Lusk and Overbeek, Editoren, *Conference on Automated Deduction (CADE)*, pp. 768–769. Springer Verlag, 1988.

[26] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[27] S. Owre, J.M. Rushby, N. Shankar und M.K. Srivas. A tutorial on using PVS for hardware verification. In T. Kropf and R. Kumar, Editoren, *Conference on Theorem Provers in Circuit Design (TPCD)*, volume 901 of *Lecture Notes in Computer Science*, pp. 258–279, Bad Herrenalb, Germany, 1994. Springer Verlag.

[28] W.A. Hunt. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460, 1989.

[29] B.T. Graham. *The SECD Microprocessor: A Verification Case Study*. Kluwer Academic Publishers, Boston/Dordrecht/London, 1992.

[30] P.J. Windley. Microprocessor verification. *Higher Order Logic Theorem Proving and its Applications*, pp. 32–37. IEEE Press, 1991.

[31] S. Tahar. *Eine Methode zur formalen Verifikation von RISC-Prozessoren*. Dissertation, Universität Karlsruhe, Institut für Rechnerentwurf und Fehlertoleranz, 1994.

[32] R. Kumar, K. Schneider und T. Kropf. Structuring and Automating Hardware Proofs in a Higher-Order Theorem-Proving Environment. *International Journal of Formal Methods in System Design*, pp. 165–230, 1993.

[33] G. Stålmarck and M. Säflund. Modelling and verifying systems and software in propositional logic. In B.K. Daniels, Editor, *Safety of Computer Control Systems (SAFECOMP)*, pp. 31–36. Pergamon Press, Oxford, 1990.

[34] K. Schneider. CTL and equivalent sublanguages of CTL*. In C. Delgado Kloos, Editor, *IFIP Conference on Computer Hardware Description Languages and their Applications (CHDL)*, pp. 40–59, 1997. IFIP, Chapman and Hall.

[35] J. Ruf and T. Kropf. A new algorithm for discrete timed symbolic model checking. In O. Maler, Editor, *Hybrid and Real-Time Systems*, pp. 18–32, Grenoble, France, 1997. Springer Verlag, LNCS 1201.

[36] K. Schneider. Model checking on product structures. In G.C. Gopalakrishnan and P.J. Windley, Editoren, *Formal Methods in Computer-Aided Design*, *Lecture Notes in Computer Science*, Palo Alto, CA, 1998. Springer Verlag.