**Informatik Spektrum manuscript No.**
(will be inserted by the editor)

# Model Checking - Grundlagen und Praxiserfahrungen

**Ralf Buschermöhle[1], Mark Brörkens[1], Ingo Brückner[1], Werner Damm[2], Wilhelm Hasselbring[3], Bernhard Josko[1], Christoph Schulte[1], Thomas Wolf[1]**

[1] Oldenburger Forschungs- und Entwicklungsinstitut für Informatik-Werkzeuge und -Systeme (OFFIS)
Bereich Sicherheitskritische Systeme
Escherweg 2, D-26121 Oldenburg
e-mail: {buschermoehle|broerkens|brueckner|josko|christoph.schulte|wolf}@offis.de

[2] Carl von Ossietzky Universität Oldenburg
Fakultät II Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik
Abteilung Sicherheitskritische Eingebettete Systeme
Escherweg 2, D-26121 Oldenburg
e-mail: damm@informatik.uni-oldenburg.de

[3] Carl von Ossietzky Universität Oldenburg
Fakultät II Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik
Abteilung Software Engineering
Escherweg 2, D-26121 Oldenburg
e-mail: hasselbring@informatik.uni-oldenburg.de

The date of receipt and acceptance will be inserted by the editor

## 1 Einführung

Mikroprozessoren werden neben ihrer Verwendung in Computern immer häufiger auch in technischen Geräten wie medizinischen Systemen, Kraftfahrzeugen, Telekommunikationsgeräten, industriellen Anlagen und alltäglichen Konsumgütern wie Fernbedienungen, Waschmaschinen und Küchengeräten eingesetzt. Computersysteme, die in technische Systeme integriert sind, bezeichnet man als „eingebettete Systeme“. Im Gegensatz zu herkömmlichen Systemen zeichnen sie sich oftmals durch strengere nichtfunktionale Anforderungen aus, wie harte Realzeitanforderungen, einen geringen Energieverbrauch oder eine geringe Störanfälligkeit hinsichtlich elektromagnetischer Felder. Die Entwicklung eingebetteter Systeme wird u.a. aufgrund immer höherer Anforderungen, einer wachsenden Anzahl von Komponenten und einer kontinuierlich steigenden Entwurfstiefe immer komplexer [15]. Die damit verbundene Beherrschung ihrer funktionalen und nichtfunktionalen Korrektheit mit gängigen Verfahren der Fehlersuche wie Reviews, Simulations- und Testverfahren stößt immer öfter an ihre Grenzen, da diese Verfahren lediglich die manuelle und damit zeit- und kostenintensive sowie fehleranfällige Überprüfung einiger Systemläufe zulassen. Eine kurzfristige Kompensation gelingt durch die Intensivierung der Anstrengungen in diesem Segment, was einen kontinuierlich ansteigenden Anteil der damit verbundenen Kosten am Entwicklungsaufwand zur Folge hat. Trotz dieser umfangreichen manuellen Tests, wurden in zahlreichen Projekten mit industriellen Partnern des Bereichs „Sicherheitskritische Systeme“ des Forschungsinstituts OFFIS [36] immer noch Fehler, die vorwiegend durch lange Testsequenzen erreichbar waren, nachgewiesen. Dies spiegelt sich auch in zahlreichen Rückrufaktionen und Pressemitteilungen wieder [43,44]. Da die Sicherstellung der Korrektheit von Hard- und Software insbesondere im Bereich sicherheitskritischer Systeme das zentral zu lösende Problem darstellt, werden in Prozessmodellen (z.B. V-Modell [48,46]) und Standards (z.B. DO-178B [40]) bereits seit geraumer Zeit formale Nachweise der Korrektheit gefordert. Grundlage dafür ist die Erstellung eines Systemmodells sowie die Formulierung der Anforderungen auf Basis formaler Notationen mit eindeutiger Semantik. Durch mathematische

Verfahren kann nachgewiesen werden, dass das Modell die spezifizierten Anforderungen erfüllt. Im Gegensatz zu herkömmlichen Testverfahren sind formale Korrektheitsnachweise vollständig, d.h. beim Testen kann nur eine begrenzte Auswahl möglicher Systemläufe betrachtet werden. Wird dabei kein Fehler gefunden, kann trotzdem einer existieren. Ein formaler Korrektheitsnachweis betrachtet hingegen alle Systemläufe. Derartige Nachweise mussten bislang ebenso wie auch die Transformation bzw. Formalisierung des Systemmodells und der Spezifikation manuell durch einen Verifikationsexperten angefertigt werden. Dies bringt zwar den Vorteil der Vollständigkeit, ist aber ebenfalls zeit- und kostenintensiv sowie fehleranfällig. Zur Vermeidung dieser Nachteile müssen Automatismen entwickelt werden, die bei der Konstruktion der Beweise unterstützen bzw. diese vollständig rechnergestützt auf Basis der im Entwicklungsprozess entstehenden Zwischenergebnisse anfertigen können. Ein aussichtsreicher Ansatz zur Lösung dieser Problematik ist das sog. „Model Checking“ [14], bei dem es sich um ein automatisiertes Verfahren zur Verifikation von endlichen zustandsbasierten nebenläufigen Systemen handelt. Die Anfertigung des mathematischen Nachweises geschieht vollständig durch einen Rechner. Sollte die nachzuweisende Eigenschaft verletzt werden, so kann dem Entwickler (bspw. in Form einer computergestützten Debugsitzung) automatisch der Fehler samt einem zu ihm führenden Systemablauf präsentiert werden.

Das bislang nicht abschließend gelöste primäre Problem des Model Checking ist die sog. „Zustandsexplosion“. Darunter versteht man das exponentielle Wachstum der Anzahl der zu untersuchenden Systemzustände bei einer „naiven“ Verifikation. Es existieren jedoch bereits zahlreiche Ansätze aus der Forschung, um dieses Problem wirkungsvoll anzugehen, so dass sich mittlerweile viele praktisch relevante Modelle erfolgreich verifizieren lassen [8].

Die Anwendung von Verifikationsmethoden ist umso sinnvoller, je früher im Entwicklungsprozess sie stattfindet, um resultierende Änderungskosten möglichst gering zu halten. Da Modelle früh im Entwicklungsprozess erstellt werden und zudem einen inhärent hohen Abstraktionsgrad im Sinne einer Plattformunabhängigkeit aufweisen, bietet es sich geradezu an, diese zu verifizieren und anschließend im Zuge einer „Sichtung“ des Entwicklungsraumes hinsichtlich der nichtfunktionalen Anforderungen an verschiedene Plattformen (z.B. Programmiersprachen) anzupassen. Diese Vorgehensweise spiegelt sich u.a. in den aktuellen Spezifikationen der Object Management Group (OMG [35]) wieder, wie z.B. der Model Driven Architecture, die den universellen Einsatz der Modellierungen fokussiert und in der Version 2.0 der Unified Modeling Language (UML [45]), in der die Ausführbarkeit der Modelle im Mittelpunkt steht. Zur Verdeutlichung des Potenzials dieser Methoden wird in den folgenden Abschnitten auf Basis einer (formalen) Systemmodellierung (Abschnitt 2) und Spezifikation temporaler Eigenschaften (Abschnitt 3) das Verfahren des Model Checkings nebst einiger Optimierungsansätze erläutert (Abschnitt 4) und durch Praxiserfahrungen seitens der Industrie (Abschnitt 6) abgerundet.

## 2 Formale Systemmodellierung

Typischerweise fallen eingebettete Systeme in die Klasse der sog. „reaktiven Systeme“ [30]. Diese terminieren nicht und sind daher nur eingeschränkt aufgrund ihres Ein- und Ausgabeverhaltens beschreibbar. Aus diesem Grund ist es i.d.R. notwendig, sich ein Bild über die internen Systemabläufe zu verschaffen, was auf Basis sog. „Zustände“ geschieht. Ein Zustand entspricht einer Momentaufnahme, die alle Variablen des reaktiven Systems mit ihren Wertebelegungen zu einem bestimmten Zeitpunkt enthält. Eine Zustandsänderung (Transition) wird jeweils durch einen Ausgangszustand und einen Folgezustand charakterisiert. Auf diese Weise lassen sich Berechnungen reaktiver Systeme als Transitionsfolgen beschreiben. Die Grundlage einer Systemmodellierung ist eine formale Notation, mit deren Hilfe sich die verwendeten Modellierungskonstrukte eindeutig interpretieren lassen. Ein gängiger Vertreter zur formalen Modellierung reaktiver Systeme sind sog. „Kripke-Strukturen“ [28], eine Form zustandsbasierter Transitionsgraphen.

**Definition 1** *Sei* $\mathbf{A}$ *eine Menge von atomaren Aussagen. Dann ist eine Kripke-Struktur* $M$ *über* $\mathbf{A}$ *ein Tupel* $M = (S, S_0, R, L)$ *mit einer endliche Menge von Zuständen* $S$*, einer Menge von Anfangszuständen* $S_0 \subseteq S$*, einer totalen Transitionsrelation* $R \subseteq S \times S$ *(d.h. für jeden Zustand* $s \in S$ *existiert eine Folgezustand* $s'$ *in* $S$ *mit* $R(s, s')$*) sowie einer Funktion* $L : S \rightarrow 2^{\mathbf{A}}$*, die jeden Zustand auf eine Menge von zugehörigen gültigen atomaren Aussagen abbildet.*

Insbesondere die Definition der totalen Transitionsrelation kann mit viel Aufwand verbunden sein, da alle Übergänge einzeln aufgezählt werden müssen. Deshalb bedient man sich i.d.R. anderer Notationen, z.B. der Prädikatenlogik, um die Berechnung des Folgezustands zu definieren und daraus automatisch die Transitionsrelation

abzuleiten. Das Bindeglied stellt in diesem Fall die Funktion $L$ dar, wobei dann mit Hilfe der Prädikatenlogik korrekte Aussagen formuliert werden, z.B. hinsichtlich der Transitionsrelation zur Berechnung des Folgezustands.

Um die Anwendung modellbasierter Verifikationsverfahren zu veranschaulichen, wird eine Klimasteuerung betrachtet, Diese regelt die Zimmertemperatur auf einen angenehmen Wert und kühlt bei kühlt bzw. heizt, wenn es zu warm bzw. zu kalt wird. Zur Darstellung des Modells der Klimaanlage benutzen wir ein stark vereinfachtes UML-Diagramm. Die Anforderungen an die Klimasteuerung regeln die Reaktionen auf die von dem umgebenden System in jedem Schritt übermittelten Temperaturwerte, jeweils in drei Ereignissen, auf die im nächsten Schritt folgendermaßen reagiert werden soll:

ZUKALT: Heizeinheit wird aktiviert und Kühleinheit wird deaktiviert.
ANGENEHM: Heiz- und Kühleinheit werden deaktiviert.
ZUWARM: Heizeinheit wird deaktiviert, Kühleinheit wird aktiviert.

Abbildung 1 zeigt das Klassen- und Zustandsdiagramm der Klimasteuerung. Das Klassendiagramm enthält die beiden booleschen Attribute kuehlen und heizen sowie die zu empfangenden Ereignisse in Form der Operationen ZUKALT, ANGENEHM und ZUWARM.

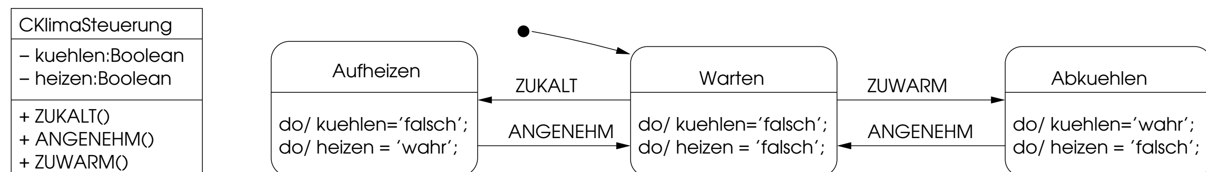


**Abbildung 1** UML-Klassen- und Zustandsdiagramm „Klimasteuerung“

Das Zustandsdiagramm besteht aus den drei Zuständen Aufheizen, Warten und Abkuehlen, wobei Warten der Startzustand ist. Die Bedingungen für die Zustandsübergänge sind mit den zu empfangenden Ereignissen beschriftet. Die Anweisungen der „Action Scripts“ sind jeweils in den Zuständen nach dem Schlüsselwort do dargestellt. Die Heiz- bzw. Kühleinheit wird aktiviert, wenn heizen bzw. kuehlen zu wahr ausgewertet werden, ansonsten werden sie deaktiviert. Aus dem Modell lassen sich nun entsprechende prädikatenlogische Formeln extrahieren (z.B. $\neg \mathsf{kuehlen} \wedge \neg \mathsf{heizen} \wedge (\mathsf{Ereignis} = \mathsf{ZUKALT}) \wedge \neg \mathsf{kuehlen}' \wedge \mathsf{heizen}'$), die wiederum die Transitionsrelation einer Kripke-Struktur beschreiben, wobei die ungestrichenen Variablen im Ausgangszustand und die gestrichenen im Folgezustand gelten. Hierbei ist zu beachten, dass die Temperaturereignisse auf Variablenausprägungen abgebildet werden müssen, da Transitionen in Kripke-Strukturen nicht annotiert werden. Grundsätzlich ist die Anzahl der Zustände der Kripke-Struktur gleich dem Produkt der Größe der Variablenwertebereiche, was in diesem Fall zwölf Zuständen entspräche. Von diesen Zuständen sind in Abbildung 2 allerdings nur die neun erreichbaren dargestellt, da es im UML-Diagramm keinen Zustand gibt, in dem gleichzeitig geheizt und gekühlt wird.

In den durch Ellipsen symbolisierten und nummerierten Zuständen sind die jeweils geltenden Variablenbelegungen dargestellt. Zunächst ist das die Belegung der Variablen Ereignis und darunter von links nach rechts die Belegung der booleschen Variablen kuehlen und heizen, wobei *Variable* bzw. $\neg$*Variable* bedeutet, dass die Variable in dem jeweiligen Zustand zu wahr bzw. zu falsch ausgewertet wird. Des Weiteren sind die entsprechenden UML-Zustände durch gestrichelte Ellipsen gekennzeichnet. Die Startzustände 1, 2 und 3 sind mit einer Eingangskante von oben mit dem gleichen Ursprung versehen. Dies bedeutet, dass nur eine Transition schaltet und somit nur ein Folgezustand erreicht werden kann.

Zudem wurde angenommen, dass die Umgebung in jedem Schritt einen Temperaturwert übermittelt. Der Vollständigkeit halber sei angemerkt, dass man in die Kripke Struktur noch einen weiteren Zustand jeweils in einen Bereich der skizzierten UML-Zustände einfügen müsste, wenn die Umgebung nicht in jedem Schritt ein Ereignis sendet (was laut UML-Semantik und ohne die angestellte Annahme in dem Modell so wäre). Dieser Zustand würde dazu dienen, die fortschreitende Schrittzahl zu erfassen, falls kein Ereignis von der Umgebung kommt.

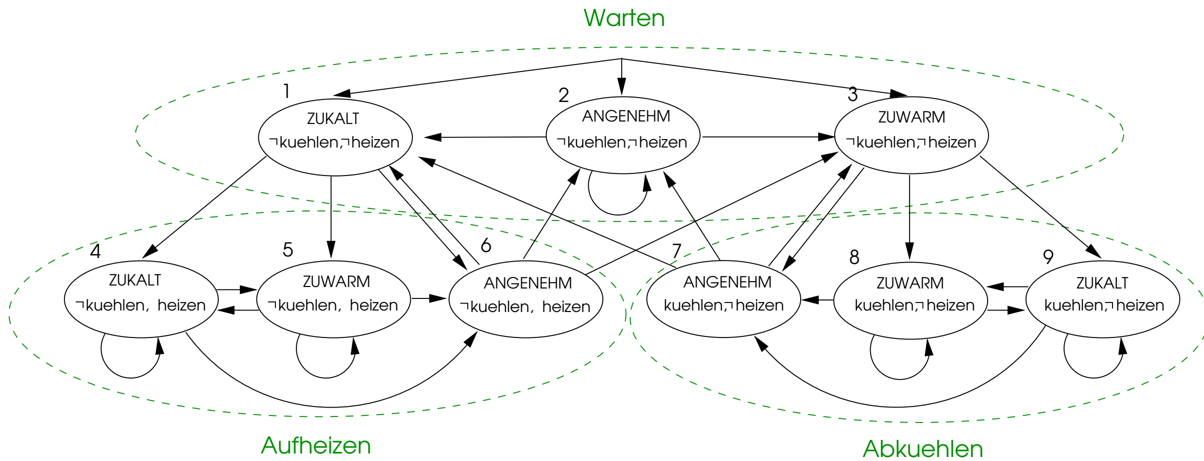


**Abbildung 2** Erreichbare Zustände der Kripke-Struktur der Klimasteuerung

## 3 Temporale Logiken

Da sich mit Hilfe der klassischen Aussagen- oder Prädikatenlogik nicht direkt zeitliche Aussagen formulieren lassen, hat man diese durch Zeitoperatoren zu sog. „Temporallogiken“ erweitert. Temporale Logiken lassen sich derart interpretieren, dass Zeit als Folge von Schritten in einem Berechnungsbaum einer Kripke Struktur betrachtet wird. Dieser kann durch das Auffalten einer Kripke-Struktur in alle von ihr möglichen Berechnungen gewonnen werden. In Abbildung 3 ist ein solcher Berechnungsbaum für die Kripke-Struktur aus Abbildung 2 dargestellt, wobei die Nummerierung der Knoten den Zustandsnummern der Kripke-Struktur entspricht und in diesem Fall vom Startzustand 1 ausgegangen wurde.

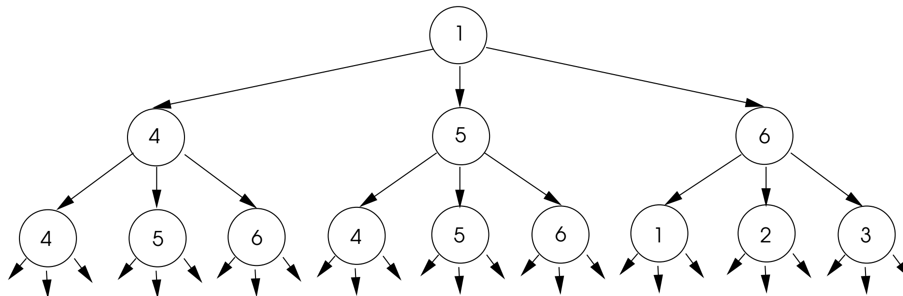


**Abbildung 3** Berechnungsbaum der berechenbaren Zustände zur Kripke-Struktur aus Abbildung 2

Temporallogiken lassen sich grundsätzlich dahingehend klassifizieren, ob sie Verzweigungen im zugrundeliegenden Berechnungsbaum betrachten, wie z.B. bei der „Computation Tree Logic“ (CTL [13]), oder ob sich mit ihrem Vokabular lediglich lineare Berechnungsabläufe ausdrücken lassen, wie z.B. mit der „Linear Temporal Logic“ (LTL [38]). Im Folgenden wird CTL zur Spezifikation von Systemeigenschaften genutzt.

In CTL sind prädikatenlogische Ausdrücke um die in Tabelle 1 dargestellten temporalen Operatoren sowie um zwei „Pfadquantoren“ erweitert. Die Pfadquantoren geben an, ob eine darauffolgende Aussage für alle ausgehenden Pfade des betrachteten Knotens im Berechnungsbaum gilt (**A**) oder nur für mindestens einen (**E**). In der Syntax der CTL (vgl. [14], S. 28) sind nur Formeln enthalten, in denen jedem temporalen Operator einer der beiden Pfadquantoren unmittelbar vorangeht. Zur Veranschaulichung sind in Abbildung 4 zwei Berechnungsbäume dargestellt, in denen die CTL-Formeln **EG** $f$ bzw. **AF** $f$ erfüllt sind. Die schwarz ausgefüllten Kreise repräsentieren Zustände, in die Pfadformel $f$ erfüllt ist.

Temporallogische Formeln bieten eine große Mächtigkeit zur Spezifikation temporaler Eigenschaften von formalen Systemen. Ihre Anwendungsfreundlichkeit ist jedoch, je nach verwendeter Notation zur Systembeschreibung,

| Operator | gelesen | (natürlichsprachliche) Bedeutung |
|---|---|---|
| **X** $f$ | neXt time | Im nächsten Zustand gilt $f$. |
| **F** $f$ | in the Future | In mindestens einem Zustand auf dem Pfad gilt $f$. |
| **G** $f$ | Globally | In allen Zuständen auf dem Pfad gilt $f$. |
| $f$ **U** $g$ | Until | Mindestens ein Zustand auf dem Pfad erfüllt $g$ und alle vorherigen Zustände erfüllen $f$. |
| $f$ **R** $g$ | Release | Genau wie „Until“, allerdings sind die Gültigkeiten von $f$ und $g$ vertauscht. |

**Tabelle 1** Temporale Operatoren von CTL (mit den Pfadformeln $f$ und $g$)

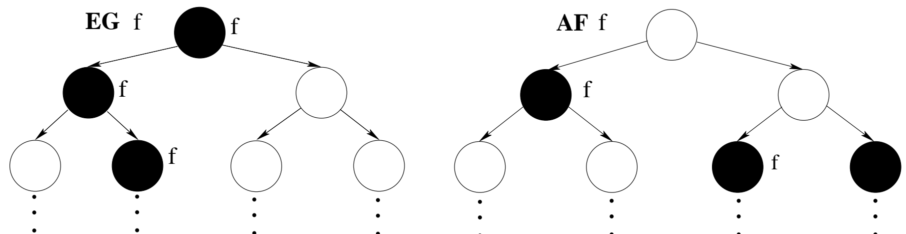


**Abbildung 4** Visualisierung von **EG** $f$ und **AF** $f$

verbesserungswürdig, da die Systemdarstellung nicht zwangsläufig den aufgefalteten Berechnungsbaum erkennen lässt. Zur Lösung dieser Problematik existieren verschiedene Herangehensweisen, zu denen beispielsweise die Entwicklung sog. „Pattern“ für häufig verwendete temporallogische Formeln oder auch verschiedene grafische Darstellungsformen wie symbolische Zeitdiagramme, sog. „Symbolic Timing Diagrams“ (STD [17]) für hardwarenahe Bereiche oder sog. „Life Sequence Charts“ (LSC [16]) gehören, die sich insbesondere zur Darstellung von Eigenschaften objektorientierter Systeme eignen, da sie an die bekannten Message Sequence Charts (MSCs [41]) angelehnt sind. Im Gegensatz zu MSCs besitzen LSCs allerdings eine größere Ausdruckskraft, da sie zwischen möglichem (existenziellem) und notwendigem (universellem) Verhalten differenzieren.

## 4 Model Checking

Die Idee des Model Checking ist im Prinzip einfach. Auf Basis einer Kripke-Struktur $M = (S, S_0, R, L)$ und einer geforderten temporalen Formel $f$ werden alle Startzustände gesucht, die $f$ erfüllen, d.h. gesucht wird die Menge $\{s \in S_0 | M, s \models f\}$ (für die formale Definition von '$\models$' vgl. [14]). Für die zu verifizierende Kripke-Struktur betrachten wir das Modell aus Abbildung 2 und für die dagegen zu verifizierende Eigenschaft die Anforderung „Jeweils einen Schritt nach dem Senden des Ereignisses ZUWARM ist die Kühleinheit aktiviert und die Heizeinheit deaktiviert“ (siehe Abschnitt 3). Mit Hilfe des bereits vorgestellten CTL Vokabulars lässt sich dies folgendermaßen formalisieren: $\mathbf{AG}((\mathsf{Ereignis} = \mathsf{ZUWARM}) \rightarrow \mathbf{AX}(\mathsf{kuehlen} \wedge \neg \mathsf{heizen}))$.

Model Checking wird nun basierend auf dem CTL Model Checking Verfahren aus [14] erläutert. Der zugrundeliegende Algorithmus basiert auf der Anwendung einer Beschriftungsfunktion label(s), die in mehreren aufeinanderfolgenden Ausführungsschritten jeden Zustand $s \in S$ mit gültigen Teilformeln von $f$ beschriftet. Im ersten Schritt der Ausführung sind dies die in den jeweiligen Zuständen gültigen atomaren Aussagen. Somit ist label(s) = L(s). In den anschließenden Schritten wird dann sukzessive die nächst größere Teilformel aus der zu prüfenden Eigenschaft verarbeitet. Somit beschriftet der $i$-te Ausführungsschritt die Zustände der Kripke-Struktur mit den CTL-Teilformeln von $f$, die eine Verschachtelungstiefe von $i - 1$ aufweisen. Bei Terminierung des Algorithmus gilt: $M, s \models f$ genau dann, wenn $f \in$ label(s). In Tabelle 2 sind die zu prüfenden Teilformeln und die Zustände der Kripke-Struktur aus Abbildung 2 gegenübergestellt, in denen sie gültig sind.

In der Tabelle ausgelassen ist lediglich die Implikation von $f$, die verlangt, dass die Zustände, in denen $\mathsf{Ereignis} = \mathsf{ZUWARM}$ gilt, eine Teilmenge der Zustände ist, die $\mathbf{AX}(\mathsf{kuehlen} \wedge \neg \mathsf{heizen})$ erfüllen. Da dies jedoch nicht der Fall ist, schlägt der Nachweis von $f$ fehl. Dies liegt daran, dass von Zustand 5 (UML-Zustand Aufheizen) nicht wie gefordert in einen der Zustände 7, 8 oder 9 (UML-Zustand Abkuehlen) geschaltet wird. Der Fehler könnte durch eine Erweiterung des UML-Diagramms um zwei weitere Transitionen behoben werden, die direkt von Aufheizen nach Abkühlen bzw. umgekehrt schalten, wie in Abbildung 5 dargestellt. In runden Klammern ist jeweils der entsprechende Zustand der (geänderten) Kripke Struktur annotiert.

| Teilformel | Gültige Zustände der Kripke-Struktur |
|---|---|
| kuehlen | {7, 8, 9} |
| ¬heizen | {1, 2, 3, 7, 8, 9} |
| kuehlen ∧ ¬heizen | {7, 8, 9} |
| **AX**(kuehlen ∧ ¬heizen) | {3, 8, 9} |
| Ereignis = ZUWARM | {3, 5, 8} |

**Tabelle 2** Teilformeln und die Zustände der Kripke-Struktur, in denen sie gültig sind.

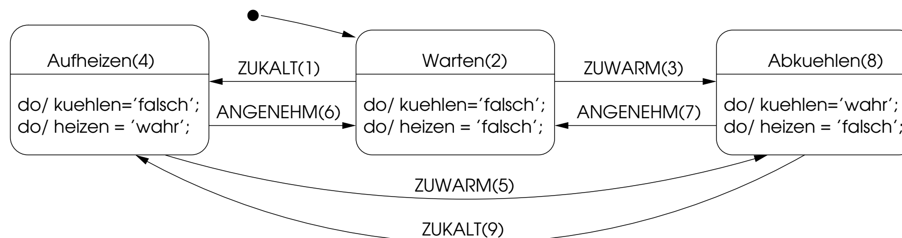


**Abbildung 5** Korrigiertes Zustandsdiagramm „Klimasteuerung“

Der zugrundeliegende Algorithmus hat die Zeitkomplexität $O(|f| * (|S| + |R|))$, wobei $|f|$ die Anzahl der zu verarbeitenden Teilformeln, $|S|$ die Anzahl der zu prüfenden Zustände und $|R|$ die Anzahl verbindender Relationen repräsentiert. $|R|$ muss in die obere Komplexitätsgrenze einbezogen werden, wenn die aktuelle Teilformel zwei zeitlich relationale Teileigenschaften, wie z.B. $f$ **U** $g$, prüft. In diesem Fall müssen zunächst alle $g$ erfüllenden Zustände und anschließend alle Zustände, die Transitionen zu den bereits beschrifteten Zuständen besitzen, beschriftet werden. Temporallogische Formeln weisen üblicherweise keine große Komplexität auf, da diese sich proportional zur Verschachtelungstiefe verhält und die Formeln schnell unübersichtlich werden können. Anders kann das bei anderen Notationen für temporale Eigenschaften aussehen. Bei der Verwendung temporallogischer Formeln stammt daher das bereits erwähnte Zustandsexplosionsproblem im wesentlichen vom hinteren Teil der Komplexitätsformel $(|S| + |R|)$, der die Modellkomplexität darstellt. Diese basiert auf der Anzahl der möglichen Zustände, die eingenommen werden können, sowie auf ihren Verbindungen untereinander, wobei die Anzahl der Zustände gleich der Mächtigkeit des kartesischen Produkts der Wertebereiche aller Variablen ist. Diese Anzahl wird zwar durch die Konzentration auf die erreichbaren Zustände reduziert, allerdings liegt sie bei praxisrelevanten Modellen häufig jenseits von „praktischer“ (d.h. innerhalb einer gegeben Zeitspanne mit einer gegebener Rechnerkonstellation) oder sogar theoretischer (jeder noch so starke Rechner würde unendlich lange benötigen) Berechenbarkeitsgrenzen, sofern man das Model Checking „naiv“ anwendet. Daher wurden bereits vor geraumer Zeit Ansätze zur Zustandsraumoptimierung entwickelt [14], von denen im Folgenden einige kurz vorgestellt werden. Die damit erzielbaren Optimierungsfaktoren hängen jedoch immer wesentlich vom verwendeten Modell bzw. der Modellierungssprache und der zu verifizierenden Eigenschaft ab, weshalb über sie i.d.R. keine allgemeingültigen Aussagen gemacht werden.

## 5 Lösungsansätze für das Problem der Zustandsexplosion

Eine Klasse von Optimierungen betrifft das sog. *symbolische Model Checking*, bei dem man Systemrepräsentationen nutzt, die im Vergleich zu Kripke-Strukturen viele Modelle vom Rechner schneller manipulieren lassen und beim Model Checking weniger Speicher verbrauchen, z.B. sog. „Ordered Binary Decision Diagrams“ (OBDDs [9]). Alle Elemente der Transitionsrelation werden auf OBDDs abgebildet. Es sind azyklische Graphen über einer linear geordneten Menge von booleschen Variablen, die als Nichtterminalsymbole innerhalb des Graphen fungieren. Seine Terminalsymbole sind jeweils mit 0 oder 1 beschriftet und jeder Nichtterminalknoten hat zwei ausgehende gerichtete Kanten, eine mit 0 und eine mit 1 beschriftet. Die Reihenfolge der Beschriftungen auf jedem Pfad gemäß der Kantenrichtung ist strikt steigend hinsichtlich der gewählten Ordnung der Variablenmenge. Jeder OBDD-Knoten repräsentiert einen booleschen Ausdruck $O_v$. Jeder Terminalknoten repräsentiert den konstanten Wert seiner Beschriftungen, d.h. entweder 1 oder 0. Jeder Nichtterminalknoten $v$ mit der Beschriftung $a_v$, dessen Nachfolger entlang der 1- bzw. 0-Kante die Knoten $u$ bzw. $w$ sind, definiert den booleschen Ausdruck

$O_v := (a_v \wedge O_u) \vee (\neg a_v \wedge O_w)$, bei dem es sich um den Teilausdruck des gesamten OBDDs handelt, den der vom Knoten $v$ ausgehende Teil-OBDD repräsentiert.

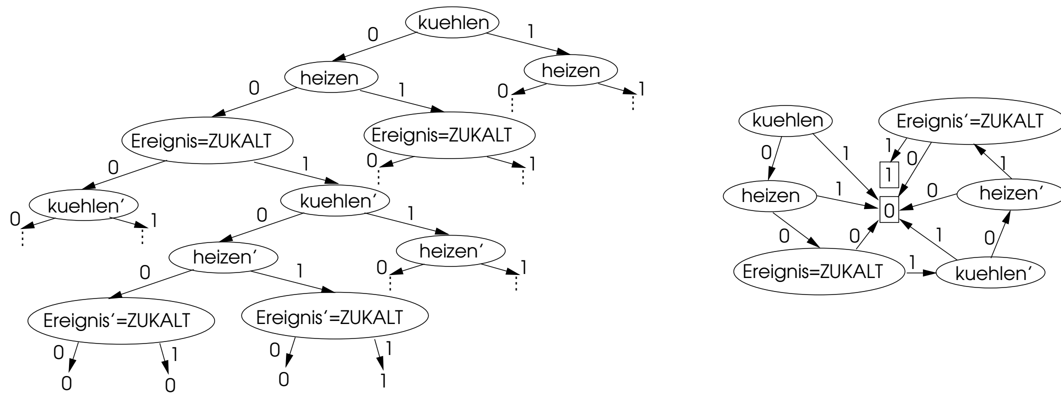


**Abbildung 6** Binärer Entscheidungsbaum und zugehöriges OBDD eines Elements der Transitionsrelation

Abbildung 6 zeigt einen Ausschnitt des binären Entscheidungsbaums und des zugehörigen OBDDs des Elements ¬kuehlen∧¬heizen∧(Ereignis=ZUKALT)∧¬kuehlen'∧heizen'∧(Ereignis'=ZUKALT) der Transitionsrelation von der Kripke-Struktur der Klimasteuerung. Da sich die Anzahl der Knoten auf jeder Ebene im Baum verdoppelt, hat der vollständige Baum 127 Nichtterminal- und 128 Terminalknoten. Aus diesem Grund und da die gekürzten Teilbäume alle zu Null ausgewertet werden, zeigt der Baum nur den Pfad, den das Element der Transitionsrelation beschreibt. In analoger Weise reduziert das OBDD den Entscheidungsbaum. Problematisch sind OBDDs, die wenig Redundanzen aufweisen und dementsprechend nur wenig Reduktionen zulassen. Zudem ist die gewählte Reihenfolge der in dem OBDD kodierten Binärvariablen ein weiterer Einflussfaktor für seine Größe, da sie die anwendbaren Optimierungen beeinflussen kann.

Die Bezeichnung „symbolisches Model Checking“ stammt daher, dass das Model Checking rein syntaktisch auf den OBDDs in Form von rückwärtsgerichteten Fixpunktberechnungen durchgeführt wird. Der Vorteil des symbolischen Model Checkings liegt in seiner „Gedächtnislosigkeit“, da man sich im Vergleich zu Kripke-Strukturen nicht merken muss, welche Zustände mit welchen Beschriftungen versehen worden sind. Dadurch fällt u.U. der zur Verifikation notwendige Speicherplatz wesentlich kleiner aus. 1987 wurde die Anwendung von OBDDs erstmals von Ken McMillan, der zum damaligen Zeitpunkt ein Student an der Carnegie Mellon Universität war, zur Verifikation großer Systeme vorgeschlagen [32]. Mit Hilfe dieser Repräsentation war er in der Lage, Modelle mit mehr als $10^{20}$ Zuständen [12] zu verifizieren. Diese Grenze konnte mittlerweile durch zusätzliche Optimierungen wesentlich erhöht werden, so dass der Exponent in der obigen Komplexitätsangabe für viele Modelle signifikant weiter angewachsen ist [10, 11]. Da symbolisches Model Checking eine vollkommen andere Repräsentation als Kripke Strukturen besitzt und damit auch andere Systeme effizient oder nicht effizient berechnen können, spricht man in diesem Kontext bei exponentiell wachsenden Problemgrößen nicht mehr von einer Zustands- sondern von einer sog. „BDD“-Explosion.

Neben der Nutzung von effizienter manipulierbaren Systemrepräsentationen nutzt man Verfahren zur Reduktion des Zustandsraumes, die in Abhängigkeit vom Modell und der zu verifizierenden Eigenschaft angewendet werden. In diesem Zusammenhang bilden *Abstraktionstechniken* Zustandsmengen auf abstraktere Teilmengen ab, wobei die zu verifizierende Eigenschaft nicht direkt oder indirekt von einem der zu abstrahierenden Modellelemente abhängen sollte. Abstraktionen lassen sich entweder manuell oder automatisch anwenden, wobei erstere von der Kreativität und Erfahrung des Modellierers abhängen. Letztere hingegen untersuchen z.B. automatisch Berechnungsbeziehungen zwischen Variablen, um daraus Hinweise für eine Wertebereichsreduktion abzuleiten [29]; sie analysieren die Unabhängigkeit konkurrierend ausgeführter Transitionen zur Beschleunigung der Verifikation asynchroner nebenläufig ausgeführter Softwareprozesse (sog. „Partial Order Reduction“ [47]) und sie nutzen Symmetrien im Systemmodell [2], um die Anzahl der notwendigen Beschriftungen bei der Verifikation durch günstiges Kopieren der symmetrischen Teile zu reduzieren. Derartige Ansätze lassen sich unter dem Begriff „explizites“ Model Checking zusammenfassen, da abstraktes und konkretes Modell immer das gleiche Verhalten hinsichtlich der für das Modell zu verifizierenden Eigenschaft aufweisen. Im Gegensatz dazu basiert der Ansatz einer weiteren Abstraktionstechnik auf der Berechnung sog. „Überapproximationen“ des Originalmodells. Hierbei

werden Transitionen bei der Approximation auf Kosten eines größeren Zustandsraumes entfernt. Dies funktioniert immer dann, wenn es günstiger ist, eine Menge von Zuständen zu beschriften als ihre exakte Teilmenge zu bestimmen, die über Transitionen miteinander in den Berechnungen beschrieben ist. Für die nachzuweisende Eigenschaft bedeutet dies: Wenn sie für das abstrakte Modell wahr ist, dann ist sie auch wahr für das Ausgangsmodell. Sollte sie jedoch falsch sein, könnte sich bei dem Versuch der Darstellung des Gegenbeispiels zeigen, dass sich das Verhalten des abstrakten und des Ausgangsmodells unterscheidet. Ist dies der Fall, wird das abstrakte Modell in Richtung des abweichenden Verhaltens des Gegenbeispiels verfeinert. Die Berechnung der Überapproximationen geschieht durch eine Sammlung weiterer Abstraktionstechniken, z.B. dem sog. „Freeing“, das den möglicherweise beschränkten Wertebereich einer Variablen, die durch eine Systemkomponente berechnet wird, durch den vollständigen Wertebereich ihres Datentyps ersetzt. In Kombination mit der sog. „Cone of Influence“ Methode, welche die Modularität von Systemstrukturen nutzt, um hinsichtlich der Spezifikation abhängige Teile zu identifizieren, können so „Überapproximationen“ bestimmt werden, indem z.B. die Transitionen einer Systemkomponente entfernt werden und ihre Ausgangsvariablen jeweils den kompletten Wertebereich einnehmen. Ein Beispiel für einen solchen Überapproximationsansatz ist die sog. „Automatic Abstraction/Refinement Methodology“ [7]. Mit Hilfe dieser Technik konnte die Verifikation zahlreicher Modelle industrieller Anwender beschleunigt bzw. erstmalig innerhalb einer praktischen Berechenbarkeitsgrenze von 24 Stunden durchgeführt werden. Die folgende Übersicht gibt einen Eindruck von den erzielten Resultaten. Als Maß für die Komplexität der Modelle sind jeweils die Zustands-Bits, die Input-Bits und die Größe der Transitionsrelation angegeben. Die Zustands- und Input-Bits korrelieren mit der Anzahl von BDD-Variablen, die notwendig waren, um das Modell zu kodieren. Die Größe der Transitionsrelation gibt Auskunft darüber, wie viele BDD-Knoten notwendig waren, um das interne Verhalten abzubilden.

Beispielsweise sollte das Modell „Radio based Signaling System“ verifiziert werden. Das Modell beschreibt ein funkgesteuertes Signalsystem für einen Bahnübergang. Es sollte u.a. geprüft werden, ob die Signale gesichert werden, nachdem die Bahnschranke geschlossen wurde (E1) und ob ein Zug nur passiert, wenn der Bahnübergang zuvor gesichert wurde (E2). Zur Verifikation kam ein UltraSparc-III System mit 750 MHz und 2.5 GB RAM zum Einsatz.

| **Eigenschaft** | **Z** | **I** | **B** | **E** | **T** |
|---|---|---|---|---|---|
| E1 (V) | 81 | 16 | 8615 | wahr | 5sec |
| E1 (N) | 24 | 29 | 5958 | wahr | 1.7sec |
| E2 (V) | 126 | 26 | 18603 | k.A. | >24h |
| E2 (N) | 10 | 36 | 1132 | wahr | 0.4sec |

**Tabelle 3** Ergebnisse der automatischen Abstraktion am Beispielmodell des funkgesteuerten Signalsystems (Z = Zustands-Bits, I = Input-Bits, B = BDD-Knoten, E = Ergebnis, T = Zeit, V/N = vor/nach der Optimierung)

Bei den zur Verifikation notwendigen Laufzeiten weiterer Modelle konnten ähnliche Optimierungsfaktoren erzielt werden (für eine vollständige Übersicht vgl. [7], Kapitel 8.2). Im schlechtesten Fall benötigte eine Iteration bei der Verifikation des abstrakten Modell die gleiche Zeit wie zuvor auf Basis gängiger Optimierungsverfahren. Die Ergebnisse in Tabelle 3 verdeutlichen anschaulich das Potenzial adäquater Optimierungstechniken.

## 6 Model Checking in der Praxis

Die manuelle Transformation von Modellen sowie die Anwendung von Model Checking Methoden ist zwar zur Veranschaulichung der Funktionsweisen in kleinen Beispielen durchaus adäquat; bei praktisch relevanten Modellgrößen ist sie jedoch i.d.R. kaum durchführbar. Daher wurden Werkzeuge für verschiedene Anwendungsbereiche entwickelt, die diese Aufgaben automatisch erledigen. Um deren Potenzial einschätzen zu können, wird im Folgenden über einige Erfahrungen berichtet, die insbesondere die Breite der Anwendungsmöglichkeiten des Model Checking verdeutlichen. Setzte man Model Checking Methoden anfangs nahezu ausschließlich zur Verifikation von Hardware ein ([33], Abschnitt 6.1), findet man sie seit einiger Zeit verstärkt auch in Projekten industriell relevanter Größenordnung aus dem Bereich der „reinen“ Software-Entwicklung (Abschnitt 6.2). Zudem nutzt man neuerdings immer häufiger eine Kombination aus modellbasierten Entwurfs- und Verifikationsmethoden (Abschnitt 6.3), um Fehler bereits früh im Entwicklungsprozess zu erkennen, und somit aufwendige Änderungsarbeiten später in der Entwicklung zu vermeiden.

### *6.1 Einsatz von Tools zur formalen Verifikation bei Motorola*

In den letzten Jahren hat die Halbleitersparte von Motorola im Entwicklungsprozess die Verwendung formaler Methoden eingeführt, um die Fehler auf der Transistorschaltebene sowie funktionale Fehler auf der Registertransferebene (RTL) zu vermeiden. Entwickelt wurden dabei das Werkzeug Versys2 sowie eine auf CBV (Cycle-Based Verilog) basierende Werkzeug-Suite [1].

Die Entwicklung von individuellen Speicherchips nach Kundenwünschen (custom memory) erfolgt durch einen manuellen Entwurf auf der Transistorschaltebene. Auf der Transistorschaltebene wird das Layout des zu fertigenden Chips festgelegt. Zusätzlich wird ein Modell auf der Registertransferebene (RTL) ebenfalls manuell entwickelt. Die RTL-Sicht ist eine funktionale Beschreibung mit Hilfe von Registern und logischen Schaltungen. Diese Sicht wird für die funktionale Verifikation des restlichen Entwurfs verwendet. Mit Hilfe von Versys2 ist es möglich die Anforderungen, die aus der Registertransferebene automatisch extrahiert werden können, gegen die Transistorschaltung zu verifizieren. Versys2 ist ein symbolischer Simulator. Es simuliert das Schaltungsmodell des Transistorschaltbildes und verifiziert die Annahmen und Bedingungen, die aus der Registertransferebene extrahiert wurden. Motorola hat Versys2 in den Entwurfsprozess für Speicherchips eingeführt und hat damit eine Vielzahl an Diskrepanzen zwischen der Registertransferebene und den zugehörigen Transistorschaltbildern finden können.

Die auf der Spezifikationssprache Cycle-based Verilog (CBV) basierende Werkzeug-Suite dient der funktionalen Verifikation auf der RTL-Ebene. Motorola setzt die Werkzeugsuite von der unabhängigen Verifikation kleiner individueller Entwurfsblöcke bis hin zur kompletten Chip-Simulation ein. Die Spezifikationssprache CBV, auf der die Werkzeug-Suite basiert, ist eine temporallogische Spezifikationssprache für die Verifikation des Hardware-Entwurfs und wurde insbesondere für Entwickler logischer Schaltungen entwickelt, die keine akademische Ausbildung haben. Die Sprache ähnelt zudem weniger einer temporallogischen Spezifikationssprache als eher einer Programmiersprache und soll bereits nach einer kurzen Einarbeitungsphase die Erstellung komplexer Spezifikationen ermöglichen.

CBV wird verwendet, um zulässige Eingaben für Unterkomponenten, zu verifizierende Eigenschaften und zu erfassenden Events, die während der simulationsbasierten Verifikation aufzuzeichnen und aufzuspüren sind, zu definieren. Beispielhaft sei die folgende Codierung einer Anforderung in CBV gezeigt, welche beschreibt, dass einem Eingangssignal req innerhalb von drei Zyklen ein Signal resp folgen soll:

```
if (~ req)                      // Das Signal req soll 'low' sein,
   if +(1) : (req)              // und nach einem Takt auf 'high' wechseln.
      if +(1 to 3) : (~ resp)   // Die nächsten 3 Takte soll auf resp gewartet werden,
         0;                     // sollte das Signal nicht kommen soll abgebrochen werden.
```

### *6.2 Analyse von Software mit dem Model Checker SPIN*

Eines der bekanntesten Werkzeuge für die „Linear Temporal Logic“ (LTL) wurde an den amerikanischen Bell Laboratories entwickelt: Der Model Checker SPIN (**S**imple **P**romela **In**terpreter, wobei Promela (**Pro**cess **Me**ta **La**nguage) die Eingabesprache für SPIN ist [24,25]). Dieser war in der Lage, in zahlreichen Fallstudien Modelle von industriell relevanter Größe zu verifizieren, die regelmäßig auf den seit 1995 jährlich stattfindenden Workshops präsentiert werden[42]. Bei SPIN handelt es sich um ein Verifikationssystem, das mit dem Ziel entwickelt wurde, den Entwurf und die Verifikation asynchroner Prozesse zu unterstützen. Eine relativ junge Anwendung von SPIN existiert im Werkzeug „FeaVer“ (**Fea**ture **Ver**ification System, [26]), das ursprünglich dafür entwickelte wurde, die Anrufverarbeitungs-Software der kommerziellen Telefonanlage „PathStar Access Server“ zu überprüfen, darüber hinaus jedoch in der Lage ist, beliebigen ANSI-C-Quellcode mit Model Checking Methoden zu analysieren. Das initiale Projekt zur Analyse der Telefonanlagen-Software wurde über einen Zeitraum von 18 Monaten begleitend zur Entwicklung der Telefonanlage und in kontinuierlicher Zusammenarbeit mit den Entwicklern der Software durchgeführt, beginnend beim ursprünglichen Entwurf im Jahr 1998 bis zum kommerziellen Vertrieb des Produkts im Jahr 2000.

Die Erstellung des formalen Systemmodells erfolgt bei „FeaVer“ zu großen Teilen automatisiert, basierend auf der eigentlichen Software sowie auf einer manuell erstellten sog. „Test Harness“. Diese legt verschiedene Parameter

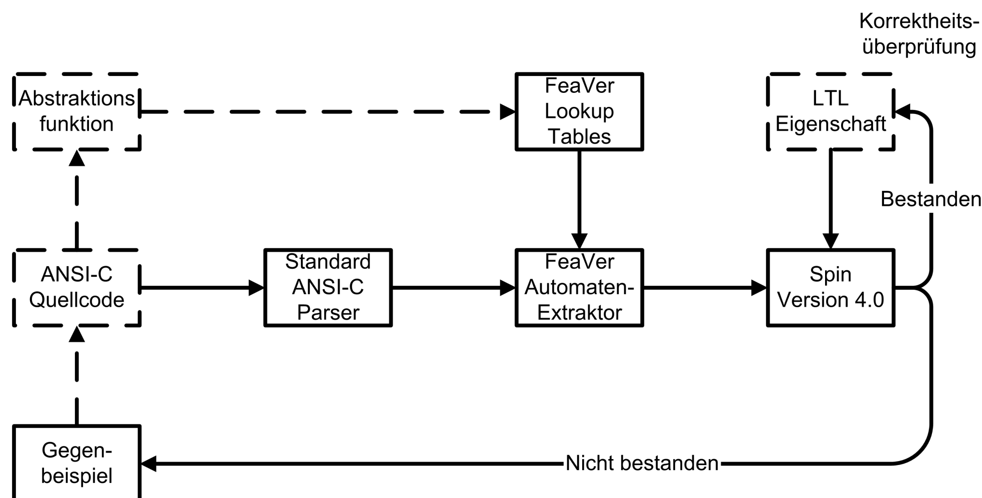


**Abbildung 7** „FeaVer“ Framework zur Extraktion von Modellen aus ANSI-C-Programmen

fest, die für die Extraktion des Promela-basierten Automaten nötig sind, der schließlich vom Model Checker SPIN analysiert werden kann. Insbesondere ist es hier möglich, Regeln für Abstraktionen zu definieren, die die Komplexität der betrachteten Programme so reduzieren, dass der Model Checker in akzeptabler Zeit Ergebnisse für LTL-Eigenschaften dieses Modell berechnen kann. Diese Abstraktionen können sehr unterschiedlich aussehen. Beispielsweise können Funktionsaufrufe dadurch abstrahiert werden, dass im Modell nur noch der Funktionsaufruf erhalten bleibt, nicht aber der in der Funktion enthaltene Code, wenn für die Verifikation beispielsweise nur die Reihenfolge der Funktionsaufrufe interessant ist. Eine andere mögliche Form der Abstraktion besteht darin, in Abhängigkeit von der Abbildung bestimmter Datentypen, deterministische durch nichtdeterministische Verzweigungen zu ersetzen.

Abbildung 7 stellt einen Ablauf einer Verifikation mit „FeaVer“ dar , wobei die gestrichelten Linien diejenigen Punkte kennzeichnen, die das Ergebnis manueller Aktivitäten sind. Dazu gehört zunächst die Erstellung des C-Codes als Ausgangsbasis , des Weiteren die Abstraktionsfunktionen, auf deren Grundlage der Automaten-Extraktor die abstrahierten Modelle erstellt, sowie die Formulierung der LTL-Eigenschaften, die schließlich durch SPIN nachgewiesen werden sollen. Obwohl bei dem Ansatz von „FeaVer“ die Notwendigkeit der zeitintensiven und fehleranfälligen manuellen Konstruktion von Verifikationsmodellen umgangen wird, sind die verbleibenden Arbeitsschritte nach wie vor recht aufwändig. Nachdem eine Konfiguration für einen Verifikationslauf mit „FeaVer“ eingerichtet wurde, stellt sich der Rest der Verifikation allerdings als verhältnismäßig einfach dar und erfordert auch bei Modifikationen des Quelltextes keinen wesentlichen zusätzlichen Aufwand, solange die zu verifizierenden Eigenschaften und die Abstraktionen nicht von den Änderungen abhängen. Für die initiale Entwicklung einer geeigneten Konfiguration für Verifikationsläufe ist jedoch nach wie vor ein nicht unerheblicher Aufwand notwendig, der aufgrund der damit verbundenen manuellen Arbeit auch selbst wiederum eine Fehlerquelle sein kann.

Die mit „FeaVer“ gewonnenen Erfahrungen zeigen jedoch, dass ein deutlich größerer Anteil von Entwurfs- und Implementierungsfehlern bei der Softwareentwicklung als mit herkömmlichen Methoden des Testens erkannt werden kann. Diese Anwendungsergebnisse von „FeaVer“ konnten in mehreren Projekten (z.B. [23]) erfolgreich bestätigt werden. Nach Auskunft von Gerard Holzmann

> „We measured ten times more errors being caught with our verification framework, compared to the traditional approach (70 vs 7). We also used ten times fewer people (2 instead of 20).“

konnte eine Effizienzsteigerung um zwei Größenordnungen bei der Fehlersuche verzeichnet werden.

### *6.3 Modellbasierte Analyse mit der OFFIS Verification Environment*

Am Forschungsinstitut OFFIS wird seit geraumer Zeit eine Umgebung zur Anbindung formaler Methoden, wie Verifikation, automatischer Testvektorgenerierung und Sicherheitsanalyse entwickelt. Für eine Anbindung an Ent-

wicklungswerkzeuge wurden bislang primär Modellierungswerkzeuge ausgewählt, um die Mächtigkeit formaler Methoden zur Fehlerfindung bereits frühestmöglich im Entwicklungsprozess zur Verfügung zu stellen. Neben dem Modellierungswerkzeug *Statemate* (*I-Logix*, [27]) wurden unter anderem *ASCET-SD* (*ETAS*, [21]), *Rhapsody* (*I-Logix*, [27]) und *SCADE* (*Esterel Technologies*, [20]) angebunden. Seitens der Model Checker können *VIS* (*University of California at Berkeley*, [49]) und der SAT-Solver *Prover CL* (*Prover Technologies*, [39]) eingesetzt werden. In experimentellem Stadium befinden sich darüber hinaus Anbindungen an die SAT-Solver *Goblin* (*Universität Oldenburg*, [22]) sowie an *zChaff* (*Princeton University*, [51]). Zudem sichert der modulare Aufbau der Umgebung die zukünftige Integration noch kommender Entwicklungswerkzeuge und Model Checker. Im Gegensatz zum vorherigen Abschnitt 6.2 kommen hierbei vollständig automatisch mittels eines Rechners durchführbare Optimierungstechniken zum Einsatz. Diese Verifikationsumgebung hat das Unternehmen OSC-Embedded Systems AG [37], ein Spin-Off von OFFIS, mittlerweile zur Marktreife weiterentwickelt und realisiert momentan zugleich eine Ankopplung an das Modellierungswerkzeug *Matlab/Simulink/Stateflow* (*The Mathworks*, [31]) unter Verwendung des Produktionscodegenerators *TargetLink* (*dSPACE*, [19]).

Die Evaluation der Verifikationsumgebung wurde bereits durch mehrere industrielle Projektpartner vorgenommen. Unter anderen begleitete der Automobilkonzern BMW die Verifikation von ASCET-SD Modellen [18]. Bei der Evaluation konnte erfolgreich eine sog. „Failsafe“-Komponente der aktiven Lenkunterstützung im neuen 5er BMW verifiziert werden. Die zu überprüfende Eigenschaft betraf das unbedingte Abschalten der Lenkunterstützung im Fehlerfall, das im System durch zwei redundante Komponenten sichergestellt wird, die bei einer korrekten Funktionalität immer die gleichen Werte ermitteln. Durch die Verifikation wurde ein Fall gefunden, in dem das System sich trotz eines Fehlers nicht abschaltete. Dieser Fehler wurde zwar auch mit manuellen Testverfahren gefunden, allerdings war der damit verbundene Aufwand ungleich höher. Ein weiterer industrieller Partner ist das Unternehmen DaimlerChrysler, das die Model Checking Lösung von OSC auf der Basis von Statemate (*I-Logix*, [27]) zu einem Kernelement seines Software-Entwicklungs-Prozesses im Bereich ’by-wire Systeme’ gemacht hat. Abschließend sei auf den Automobilkonzern Nissan verwiesen, der zur Sicherstellung der Entwurfsanforderungen den Einsatz von Model Checking Technologien bereits auf Modellebene und damit noch vor der Erstellung aufwändiger Prototypen fest im Entwurfsprozess vorsieht [34] .

Aber nicht nur im Automobil-, sondern auch im Bahn- und Luftfahrtbereich bestätigen Anwender den sinnvollen Einsatz formaler Methoden. Eine der genannten primären Schwierigkeiten besteht häufig noch in der Formalisierung der Anforderungen in entsprechenden temporalen Notationen. Daher wird bereits seit längerem an der Entwicklung von weiteren anwendungsspezifischen textuellen und visuellen Darstellungen temporaler Eigenschaften gearbeitet (vgl. Abschnitt 3).

## 7 Fazit

Model Checking ist bereits seit langem eine interessante Methode zur automatischen Anwendung formaler Verifikationstechniken. Die praktischen Anwendungsbereiche fokussierten bislang jedoch eher auf kleinere Systeme. Aufgrund aktueller Forschungsergebnisse in Kombination mit immer leistungsfähigeren Rechnern konnten die praktischen und theoretischen Berechenbarkeitsgrenzen immer weiter ausgeweitet werden, so dass mittlerweile komplexere praxisrelevante Systeme erfolgreich verifiziert werden konnten. Neben den in diesem Artikel dargestellten Ansätzen des Model Checkings werden außerdem verschiedene andere Techniken verfolgt, mit denen beispielsweise auch Systeme mit Echtzeitanforderungen behandelt werden können ([4,5]) oder mit denen es möglich ist, Modelle hybrider Systeme zu analysieren, bei denen kein endlicher Zustandsraum vorausgesetzt wird ([3]). Zusammenfassend lässt sich feststellen, dass die Entwicklungszeit eingebetteter Systeme durch die formale Verifikation verkürzt werden kann. Außerdem ermöglicht sie es, Fehler zu entdecken, die mit konventionellen Testmethoden nicht gefunden worden wären [6]. Ist zudem die nahtlose Integration in den Entwicklungsprozess gewährleistet, dürfte es nur noch eine Frage der Zeit sein, bis formale Methoden ein fester Bestandteil von Entwicklungsprozessen geworden sind.

**Danksagung:** Das diesem Artikel zugrundeliegende Vorhaben wurde mit Mitteln des Bundesministeriums für Bildung und Forschung unter dem Förderkennzeichen „01ISA02G“ („ViSEK“ - Virtuelles Software Engineering Institut) gefördert. Die Verantwortung für den Inhalt dieser Veröffentlichung liegt bei den Autoren. Weitere Informationen zu den Themen formale Methoden, modellbasierte Entwicklung und Unterstützung von Entwicklungsprozessen sind im ViSEK Portal [50] zu finden.

## 8 Zusammenfassung

Die Gewährleistung der korrekten Funktionsweise von Hard- und Software ist ein entscheidender Faktor bei der heutigen Systementwicklung. Dies trifft ganz besonders auf das Gebiet der sog. „sicherheitskritischen“ Systeme zu, bei dem ein Systemversagen Menschenleben gefährden kann. Aber auch in weniger kritischen Bereichen wird die Korrektheit der bereitgestellten Funktionalität immer wichtiger. Weiterhin erhöht sich die Komplexität der Systemfunktionalität ständig, so dass manuelle Test- und Simulationsverfahren viele Fehler nur noch unter unverhältnismäßig großem zeitlichen und personellen Aufwand aufdecken können. Vor diesem Hintergrund stellt dieser Artikel die Grundlagen des sog. „Model Checkings“ vor, einer automatisch durchführbaren, vollständigen Verifikationsmethode. Ergänzt wird diese Einführung durch einige Beispiele von Erfahrungen, die durch die Anwendung von Model Checking Werkzeugen in industriellem Umfeld gewonnen wurden.

## 9 Abstract

The correct functioning of hard- and software components is often a crucial factor in computer-based system engineering. Particularly, this is the case in the area of „safety critical“ systems, where a system failure can endanger human life. But also in less critical areas the correctness of provided functionality becomes more and more important. Furthermore the complexity of system functionality increases steadily. Therefore manual test and simulation methods can detect many errors only with inaccaptable high effort concerning time and resources. Starting from this background, this article presents the basics of „model checking“, an automatic and complete verification method. Based on this introduction, experience gained with the application of model checking tools in industrial contexts is presented and discussed.

## Literatur

1. M. S. Abadir, K. L. Albin, J. Havlicek, N. Krishnamurthy, and A. K. Martin. Formal Verification Successes at Motorola. *Formal Methods in System Design*, 22:117–123, 2003.
2. K. Ajami, S. Haddad, and J.-M. Ilié. Exploiting Symmetry in Linear Time Temporal Logic Model Checking: One Step Beyond. *Lecture Notes in Computer Science*, 1384, 1998.
3. R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736, pages 209–229. Springer Verlag, 1993.
4. R. Alur and D. Dill. A theory of timed automata. *Theoret. Comput. Sci.*, 126:283–235, 1994.
5. R. Alur and T. A. Henzinger. Logics and models of real time: A survey. In *Real Time: Theory in Practice*, volume 600, pages 74–106. Springer Verlag, 1992.
6. T. Bienmüller, J. Bohn, H. Brinkmann, U. Brockmeyer, W. Damm, H. Hungar, and P. Jansen. Verification of automotive control units. *Correct System Design*, 1710:319–341, 1999.
7. T. Bienmüller. *Reducing Complexity for the Verification of Statemate Designs*. PhD thesis, Fachbereich Informatik, Carl von Ossietzky Universität Oldenburg, 2003.
8. J. Bohn, W. Damm, J. Klose, A. Moik, and H. Wittke. Modeling and Validating Train System Applications Using Statemate and Live Sequence Charts. In *Proceedings of the Conference on Integrated Design and Process Technology (IDPT2002)*. Society for Design and Process Science, Society for Design and Process Science, 2002.
9. R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Comput.*, C-35(8):677 – 691, Aug. 1986.
10. J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic Model Checking with Partitioned Transition Relations. In A. Halaas and P.B. Denyer, editors, *International Conference on Very Large Scale Integration*, pages 49 – 58. North-Holland, Edinburgh, Scotland, 1991.
11. J. R. Burch, E. M. Clarke, D. E. Long, K. L. MacMillan, and D. L. Dill. Symbolic Model Checking for Sequential Circuit Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401 – 424, 1994.
12. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: $10^{20}$ States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1 – 33. IEEE Computer Society Press, Washington, D.C., 1990.
13. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2):244 – 263, 1986.
14. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

15. W. Damm and M. Cohen. Advanced validation techniques meet complexity challange in embedded software development. *Embedded Systems Journal*, 2001.
16. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *FMOODS'99 IFIP TC6/WG6.1*, pages 293 – 312. Third International Conference on Formal Methods for Open Object-Based Distributed Systems, Kluwer, 1999.
17. W. Damm, B. Josko, and R. Schlör. Specification and verification of VHDL-based system-level hardware design. In E. Börger, editor, *Specification and Validation Methods*, pages 331–410. Oxford University Press, 1995.
18. W. Damm, C. Schulte, M. Segelken, H. Wittke, U. Higgen, and M. Eckrich. Formale Verifikation von ASCET Modellen im Rahmen der Entwicklung der Aktivlenkung. accepted by Informatik 2003 GI-workshop automotive SW engineering & concepts, September 2003.
19. dSPACE GmbH. http://www.dspace.de/, 2003.
20. Esterel Technologies. http://www.esterel-technologies.com/, 2003.
21. ETAS. http://www.etas.de/, 2003.
22. M. Fränzle and C. Herde. Efficient SAT engines for concise logics: Accelerating proof search for zero-one linear constraint systems, 2003. in Bearbeitung.
23. P. Gluck and G. Holzmann. Using SPIN Model Checking for Flight Software Verification. In IEEE, editor, *Proc. 2002 Aerospace Conference*. Big Sky, MT, USA, March 2002.
24. G. J. Holzmann. *Design and Verification of Computer Protocols*. Prentice Hall, London, 1991.
25. G. J. Holzmann. *Tutorial: Design and Validation of Protocols*. Prentice Hall, London, 1991.
26. G. J. Holzmann. Software Analysis and Model Checking. In E. Brinksma and K. G. Larsen, editors, *Computer Aided Verification. 14th International Conference, CAV 2002. Copenhagen, Denmark, July 2002.* Springer Verlag, 2002.
27. I-Logix. http://www.ilogix.com/, 2003.
28. S. A. Kripke. Semantical considerations on modal logic. In L. Linsky, editor, *Reference and Modality*. Oxford University Press, 1971.
29. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property Preserving Abstractions for the Verification of Concurrent Systems. *Formal Methods in System Design*, 6(1):11–44, 1995.
30. Z. Manna and A. Pnueli. *Temporal Verifications of Reactive Systems*. Springer, Heidelberg, 1995.
31. The MathWorks. http://www.mathworks.com/, 2003.
32. K. L. McMillan. *Symbolic model checking - an approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, 1992.
33. C. Meinel and T. Theobald. Geordnete binäre Entscheidungsgraphen und ihre Bedeutung im rechnergestützten Entwurf hochintegrierter Schaltkreise. *Informatik Spektrum*, 20:268–275, 1997.
34. Nissan selects I-Logix' Statemate MAGNUM Tool Chain for Complete Model Based Development Process of Body Electronics Systems. http://www.ilogix.com/news/press_detail.cfm?pressrelease=2002_09_09_115242_749386pr.cfm, 2003.
35. Object Management Group. http://www.omg.org, 2003.
36. Homepage: Oldenburger Forschungs- und Enwicklungsinstitut für Informatik-Werkzeuge und Systeme. http://www.offis.de, 2003.
37. OSC - Embedded Systems AG. http://www.osc-es.de/, 2003.
38. A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46 – 57. IEEE, 1977.
39. Prover Technologies. http://www.prover.com/, 2003.
40. RTCA. *DO-178B*. RTCA, 1140 Connecticut Avenue, Northwest, Suite 1020, Washington, D.C. 20036-4001 U.S.A., 1992.
41. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language reference manual*. Addison-Wesley Longman Ltd., 1999. ISBN 0-201-30998-X.
42. Spin - Formal Verification. http://spinroot.com/spin/whatispin.html, 2003.
43. Touring Club Suisse: Rückruf von Personenwagen. www.tcs.ch/WEBTCS/Resources.nsf/(ImageName)/3229de.pdf/$FILE/3229de.pdf, 2003.
44. Thailändischer Minister in BMW gefangen. http://www.spiegel.de/panorama/0,1518,248319,00.html, 2003.
45. Unified Modeling Language. http://www.omg.org/uml, 2003.
46. UML-Repräsentation des V-Modells. http://www.visek.de/?7445, 2003.
47. A. Valmari. A stubborn attack on state explosion. In *DIMACS*, pages 25–42. ACM, 1990.
48. G. Versteegen. *Das V-Modell in der Praxis - Grundlagen, Erfahrungen, Werkzeuge*. dpunkt.verlag, 2000.
49. VIS Homepage. http://www-cad.eecs.berkeley.edu/˜vis/, 2003.
50. ViSEK Homepage. http://www.visek.de, 2003.
51. zChaff. http://www.ee.princeton.edu/˜chaff/zchaff.php, 2003.