# *Logik und Automaten: ein echtes Dreamteam*

**Henrik Björklund**
**Wim Martens**
**Nicole Schweikardt**
**Thomas Schwentick**

Neben vielen anderen Anforderungen umfasst das Kerngeschäft von Informatikerinnen und Informatikern sicherlich die Spezifikation und die Implementierung von (IT-)Systemen und die Überprüfung, dass beide einander entsprechen. Ihrem Wesen nach besteht eine Spezifikation aus einer Menge von Eigenschaften, die das System erfüllen soll, ohne jedoch das System vollständig zu beschreiben. Es wäre ideal, wenn die Korrektheit einer Implementierung, also ihre Übereinstimmung mit der Spezifikation, *automatisch* überprüft werden könnte. Im Allgemeinen ist dies jedoch ein, gelinde gesagt, schwieriges Problem. Für allgemeine Softwaresysteme ist es aus theoretischer Sicht unlösbar (formal: *unentscheidbar*)[1], sogar schon für sehr einfache Eigenschaften. Auch wenn das Problem im Allgemeinen unlösbar ist, gibt es doch Bereiche, in denen es möglich ist, den Umgang mit Spezifikationen zu automatisieren. Wir wollen dies hier anhand einiger Beispiele aus der automatischen Verifikation und dem Umgang mit XML-Dokumenten illustrieren.

**Logiken und Automaten.** In diesem Artikel nennen wir eine Sprache zur Spezifikation von Eigenschaften eines Systems oder eines anderen „Informatikobjektes" eine *Logik* und ein ausführbares Modell (Algorithmus, Programm) zum Test einer Eigenschaft einen *Automaten*. Logiken beschreiben Zusammenhänge auf einer höheren Abstraktionsebene, und ihre Aussagen beschreiben Eigenschaften eher deklarativ. Automaten sind hingegen ausführbare Modelle und deshalb systemnäher und eher operationell. Ein sehr einfaches Beispiel sind reguläre Ausdrücke als Beschreibungssprache für Eigenschaften von Zeichenketten (als sehr einfache „Systeme") und endliche Automaten als zugehöriges ausführbares Modell.

**Logiken im Sinne der Informatik.** Die hier beschriebene Auffassung davon, was eine Logik ist, unterscheidet sich deutlich von der Sichtweise der Philosophie oder der Grundlagen der Mathematik. Sie ist eher pragmatisch und typisch für die Informatik. Wie das Beispiel regulärer Ausdrücke zeigt, muss eine Logik im Sinne der Informatik keineswegs wie eine „klassische" Logik aussehen.[2] Allerdings müssen wir auch in der Informatik fordern, dass klar beschrieben ist, wie die Ausdrücke einer gegebenen Logik gebildet werden (die *Syntax* der Logik) und welche Bedeutung sie haben sollen, wann also ein System oder Objekt $S$ die durch einen Ausdruck $f$ beschriebene Eigenschaft hat (die *Semantik* der Logik).

[1] Die dahinter steckende allgemeine Aussage ist als *Satz von Rice* der einen oder dem anderen vielleicht noch aus dem Informatikstudium geläufig.

[2] Wir werden jedoch noch sehen, dass es durchaus enge Beziehungen zwischen regulären Ausdrücken und klassischen Logiken gibt.

DOI 10.1007/s00287-010-0465-z


Henrik Björklund
Umeå Universitet, Dep. of Computing Science,
901 87 Umeå, Schweden
E-Mail: henrikb@cs.umu.se

Wim Martens · Thomas Schwentick
Lehrstuhl Informatik 1 – Logik in der Informatik,
Technische Universität Dortmund
44227 Dortmund, Deutschland
E-Mail: {wim.martens, thomas.schwentick}@udo.edu

Nicole Schweikardt
Goethe-Universität Frankfurt am Main, Institut für Informatik,
Postfach 11 19 32, 60054 Frankfurt am Main, Deutschland
E-Mail: schweika@informatik.uni-frankfurt.de

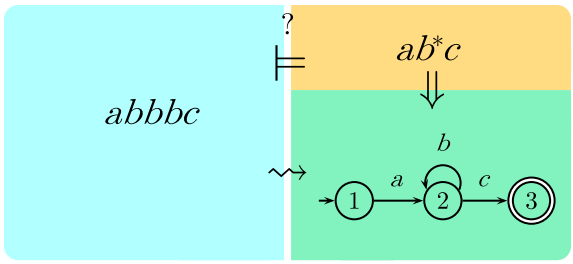



***Abb. 1 Ein regulärer Ausdruck und ein zugehöriger endlicher Automat mit 3 Zuständen. Der Automat startet in Zustand 1 und geht beim Lesen eines $a$ in Zustand 2 über. Ist er am Ende der Eingabe im akzeptierenden Zustand 3, so passt das Wort zum Muster $ab^*c$. Alle nicht eingezeichneten Übergänge führen in einen ablehnenden Zustand***

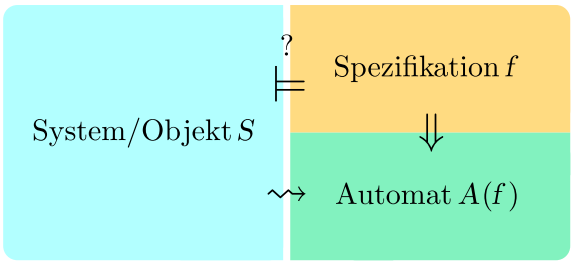



***Abb. 2 Illustration des allgemeinen Zusammenhangs zwischen Systemen/Objekten, Spezifikationen und Automaten. Man möchte wissen, ob $S \models f$ gilt oder nicht. Dazu wird $f$ automatisch in einen Automat $A(f)$ umgewandelt, sodass $S$ genau dann von $A(f)$ akzeptiert wird, wenn $S \models f$. Insgesamt liefert dies eine Methode zur automatischen Überprüfung, ob $S$ die Spezifikation $f$ erfüllt***

Wie in der mathematischen Logik drücken wir den Fall, dass $S$ die durch $f$ beschriebene Eigenschaft hat, durch $S \models f$ aus und nennen $S$ ein *Modell* von $f$. Ein sehr einfaches Beispiel dafür ist etwa die Aussage $abbbc \models ab^*c$, die ausdrückt, dass der String $abbbc$ dem durch den regulären Ausdruck $ab^*c$ beschriebenen Muster entspricht.

*Das Modelcheckingproblem.* Das Problem, zu testen, ob, für ein System $S$ und einen Ausdruck $f$, die Aussage $S \models f$ gilt, wird das *Modelchecking*-problem genannt. Um beispielsweise zu testen, ob der String $abbbc$ dem regulären Ausdruck $ab^*c$ genügt, können wir Letzteren in einen endlichen Automaten umwandeln und diesen mit der Eingabe $abbbc$ auswerten. Abbildung 1 illustriert diesen Zusammenhang.

*Das Erfüllbarkeitsproblem.* Oft sind wir auch daran interessiert, logische Beziehungen zwischen Ausdrücken von Spezifikationslogiken zu erkennen. Wenn die Gültigkeit eines Ausdrucks $f$ beispielsweise immer die Gültigkeit des Ausdrucks $g$ nach sich zieht, kann man sich (unter Umständen viele Millionen Mal) die Auswertung von $g$ ersparen, wenn man schon weiß, dass das System $f$ erfüllt. Um diesen allgemeinen Zusammenhang zwischen $f$ und $g$ festzustellen, genügt es, zu überprüfen, ob gleichzeitig $f$ gelten und $g$ nicht gelten kann, d. h., zu testen, ob $f \wedge \neg g$ *erfüllbar* ist.

**„Logic & Automata Engineering“.** Damit eine Logik zur Systemspezifikation verwendet werden kann, sollte die Logik ausdrucksstark sein, also möglichst alle interessanten Systemeigenschaften ausdrücken können. Andererseits sollte die Übersetzung in Automaten und die Lösung des Model Checking und des Erfüllbarkeitsproblems algorithmisch möglich sein und dies auch noch effizient. Natürlich sind dies tendenziell widerstreitende Ziele, und die Kunst, hier einen guten Kompromiss zu finden, lässt sich mit dem Stichwort „Logic & Automata Engineering“ beschreiben. Abbildung 2 illustriert den Zusammenhang zwischen Systemen oder Objekten auf der einen Seite und Spezifikationen durch logische Ausdrücke und ihre Übersetzung in Automaten auf der anderen Seite.

Wie wir im Folgenden sehen werden, verwenden verschiedene Anwendungsbereiche dabei völlig unterschiedliche Logiken, die mit der klassischen Prädikatenlogik (zumindest auf den ersten Blick) nicht viel zu tun haben. Typisch ist jedoch, dass, wie in der klassischen Logik, Ausdrücke aus einfachen Grundaussagen mit verschiedenen logischen Operatoren und „Quantoren“ zusammen gesetzt werden können. Die verwendeten Automatenmodelle ergeben sich in der Regel durch die Erweiterung von endlichen Automaten. So werden im Bereich der Verifikation zum Beispiel Automaten betrachtet, die auf unendlichen Strings operieren (sogenannte *Büchi-Automaten*), während im XML-Bereich Automaten betrachtet werden, deren Eingabe aus Bäumen anstelle von Strings besteht (sogenannte *Baumautomaten*).

## Model Checking

„Model Checking ist ein Verfahren zur vollautomatischen Verifikation einer Systembeschreibung (Modell) gegen eine Spezifikation (Formel).“ So sagt

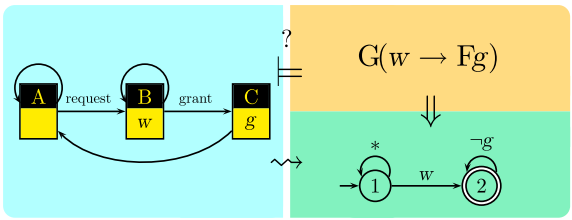



***Abb. 3 Ein Transitionssystem, ein Ausdruck $f$ und der zugehörige Automat $A(\neg f)$***

es die deutsche Wikipedia.[3] Gegeben sind die Beschreibung eines Hardware- oder Softwaresystems $S$ und eine (erwünschte oder zu vermeidende) Eigenschaft, die durch eine Formel $f$ beschrieben wird, es soll also vollautomatisch getestet werden, ob $S \models f$ gilt. Systeme werden dabei als *Transitionssysteme* (oder auch: *Kripke-Strukturen*) modelliert. Wie Automaten haben sie Zustände, in denen allerdings gewisse elementare Eigenschaften (sogenannte Propositionen) gelten können. Abbildung 3 zeigt links ein Transitionssystem, das einen simplen „Request-Grant-Zyklus" modelliert. Ein Prozess kann im „Idle"-Zustand $A$ eine Ressource anfordern und in den Wartezustand $B$ übergehen. Dort gilt die Proposition $w$ (für *warten*). Wird die Ressource zugewiesen, folgt ein Übergang in Zustand $C$, von dem aus dann wieder Zustand $A$ erreicht wird. Eine typische wünschenswerte Eigenschaft ist, dass jedem *Request* ein *Grant* folgen sollte. Das lässt sich durch den Ausdruck $\mathsf{G}(w \rightarrow \mathsf{F}g)$ beschreiben, ein Ausdruck der Logik LTL (*Linear Temporal Logic*).[4] Der Operator G (*global*) verlangt, dass das Folgende zu jedem Zeitpunkt des Laufes gilt. Die Teilformel $(w \rightarrow \mathsf{F}g)$ drückt aus, dass auf einen Zustand, in dem $w$ gilt, irgendwann in der Zukunft (F steht für *Future*) ein Zustand folgt, in dem $g$ gilt.

**Die automatenbasierte Methode.** Um nun „vollautomatisch" herauszufinden, ob $S \models f$ gilt, konstruiert die automatenbasierte Methode im ersten Schritt aus $f$ einen *Büchi-Automaten* $A(\neg f)$, der alle Berechnungen akzeptiert, die den Ausdruck $f$ *nicht* erfüllen.

Im Beispiel sind dies alle Berechnungen, in denen irgendwann ein $w$ vorkommt, dem kein $g$ folgt (siehe Abb. 3). Ein solcher Büchi-Automat für unendliche Strings (sogenannte *ω-Strings*) akzeptiert seine Eingabe, wenn er unendlich oft einen akzeptierenden Zustand durchläuft. Zu beachten ist, dass der Automat so lange im akzeptierenden Zustand 2 bleiben kann, wie er kein $g$ sieht. Im Zustand 1 kann er beim Lesen eines $w$ im Zustand 1 bleiben oder in den Zustand 2 übergehen (nichtdeterministisches Verhalten).

Nun können im zweiten Schritt das Transitionssystem $S$ und der Automat $A(\neg f)$ so miteinander verknüpft werden, dass der Produktautomat $P$ gewissermaßen alle Systemläufe von $S$ „konsumieren" kann. Eine von $P$ akzeptierte Berechnung entspricht dann einem Systemlauf, der $f$ nicht erfüllt. $S \models f$ gilt also genau dann, wenn $P$ keine einzige Berechnung akzeptiert. Dies kann in linearer Zeit in der Größe von $P$ getestet werden. Im Beispiel aus Abb. 3 stellt sich dabei heraus, dass es Systemläufe gibt, die die Eigenschaft nicht erfüllen: Sie bleiben für immer im Wartezustand $B$.

**LTL und PSL.** Die meisten Bestandteile von LTL haben wir schon kennengelernt. Zu erwähnen sind noch Ausdrücke der Form „$p$ Until $q$", die fordern, dass irgendwann in der Zukunft eine Eigenschaft $q$ gilt und bis dahin immer $p$ gilt. Die Logik LTL ist Teil einer interessanten Geschichte, die sich von der philosophischen Logik (Aristoteles, Prior) über theoretische Untersuchungen verschiedener temporaler Logiken bis zum IEEE Standard 1850 *Property Specification Language (*PSL*)* erstreckt. Diese Geschichte wird in einem äußerst lesenswerten Artikel von Moshe Vardi [25] erzählt, der in dem Sammelband *25 Years of Model Checking* [15] erschienen ist. In diesen 25 Jahren wurden mit Model Checking spektakuläre Erfolge erzielt (und viele Arbeitsplätze für Informatiker geschaffen), die meisten davon bei der automatischen Verifikation von Hardware oder Kommunikationsprotokollen. Dort hat sich das Model Checking zu einer industriellen Standardmethode entwickelt, die zur unentbehrlichen Ergänzung des herkömmlichen Testens geworden ist. Die Hauptherausforderung ist jeweils, mit der meist riesigen Anzahl von Zuständen (State Explosion) umgehen zu können. Dafür wurden ausgefeilte Methoden entwickelt. Der prinzipielle Vorteil von Hardwaresystemen ist aber, dass die Zustandsräume der zugehörigen Transitionssysteme *endlich* sind. Beim Software Model Checking können die Zustandsräume unendlich werden und

[3] Zuletzt besucht am 23.4.2010.

[4] Diese Logik heißt *linear*, weil sie verlangt, dass ein Ausdruck für alle möglichen Systemläufe gilt. Im Gegensatz dazu kann die Logik CTL (Computation Tree Logic) auch Aussagen über Verzweigungen in Systemläufen machen.

daher müssen hier ganz neue Techniken entwickelt werden.

**Klassische Logiken.** Bevor wir uns diesem Thema zuwenden, sei noch der Hinweis erlaubt, dass sich die Mächtigkeit der spezialisierten Logiken LTL und PSL durch klassische mathematische Logiken charakterisieren lässt. Erstaunlicherweise lassen sich in LTL *genau* diejenigen Eigenschaften von Systemläufen beschreiben, die sich auch in der klassischen Prädikatenlogik ausdrücken lassen. Für PSL wurde eine größere Ausdrucksstärke angestrebt: Neben anderen Erweiterungen wurde LTL um reguläre Ausdrücke erweitert und damit die Ausdrucksstärke der *monadischen Logik zweiter Stufe* (MSO) erzielt. In MSO darf nicht nur über Elemente einer Struktur quantifiziert werden ($\exists x$) sondern auch über *Mengen von Elementen*. Auf endlichen Strings lassen sich mit MSO-Ausdrücken genau die regulären Sprachen beschreiben. Dies lässt sich für $\omega$-Strings erweitern: Reguläre $\omega$-Sprachen lassen sich genau von Büchi-Automaten erkennen und von MSO-Ausdrücken beschreiben (siehe z. B. [23]). Die MSO-Logik spielt auch für XML eine wichtige Rolle, wie wir später sehen werden.

Weitere Informationen zum Thema Model Checking bieten in kurzer Form das Büchlein [3] und ausführlich das Lehrbuch [1].

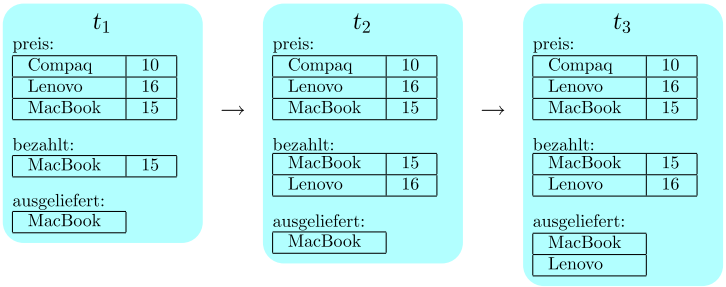
**Hardware vs. Software.** Wie wir gesehen haben, wurden mit Model Checking große Erfolge erzielt. Hauptsächlich liegen diese Erfolge im Hardwarebereich. Beim Model Checking für *Software* entstehen neue Herausforderungen. Die wichtigste ist, dass wir bei Software Model Checking meist Transitionssysteme mit *unendlichen Zustandsräumen* betrachten müssen. Während ein Hardwaresystem nur eine endliche Menge von Transistoren, Schaltkreisen usw. besitzt, hat ein Softwaresystem Variablen, Datenstrukturen etc., deren Größe wir nicht a priori kennen. Wenn wir sehr kleine oder sehr einfache Softwaresysteme analysieren wollen, können wir davon oft abstrahieren, indem wir zum Beispiel eine obere Schranke für die Größe der Datenstrukturen einführen. Wenn wir solche Einschränkungen machen, können wir auch direkt die im vorigen Abschnitt skizzierten Techniken verwenden. Leider ist es aber oft unmöglich, Einschränkungen zu finden, die nicht auch einige der *wichtigen* Eigenschaften des Systems wegabstrahieren. Daher ist Softwareanalyse gewissermaßen inhärent unendlich. Um trotzdem Softwaresysteme automatisch überprüfen zu können, sind in den vergangenen Jahren verschiedene Methoden für *Infinite State Model Checking* (Model Checking für unendliche Zustandsräume) entwickelt worden.

**Safety- und Liveness-Eigenschaften.** Zur Modellierung von Softwaresystemen werden meist *unendliche Transitionssysteme* genutzt. Diese werden natürlich nicht explizit, sondern durch symbolische Abstraktionen repräsentiert. Oft werden diese Modelle jedoch so kompliziert, dass es – selbst für sehr ausdrucksschwache Logiken – unentscheidbar ist, ob die Modelle eine gegebene logische Formel erfüllen. Daher hat sich die Forschung zum „Infinite State Model Checking“ bisher meist mit relativ einfachen (aber sehr wichtigen) *Safety-* und *Liveness-Eigenschaften* beschäftigt. Eine *Safety-Eigenschaft* garantiert, dass eine bestimmte Menge von Fehlerzuständen *nie* erreicht wird. Eine *Liveness-Eigenschaft* garantiert, dass eine bestimmte Menge von erwünschten Zuständen *immer wieder* erreicht wird.

Safety- und Liveness-Eigenschaften lassen sich durch eine *Erreichbarkeitsanalyse* bzw. *wiederholte Erreichbarkeitsanalyse* überprüfen. In LTL-Notation können Safety-Eigenschaften als $\mathsf{G}\neg f$ und Liveness-Eigenschaften als $\mathsf{GF}\, e$ ausgedrückt werden, wobei $f$ und $e$ besagen, dass ein Zustand zur Fehlermenge bzw. zur erwünschten Menge gehört.

Man beachte, dass schon das Überprüfen dieser relativ einfache Eigenschaften recht schwierig werden kann. In den vergangenen Jahren wurden verschiedene Methoden entwickelt, um die Fälle zu identifizieren, in denen die Überprüfung trotzdem praktisch durchführbar ist. Hierzu gehören beispielsweise die Theorie der aufwärts abgeschlossenen Mengen und der Wohlquasiordnungen.

**Exkurs: LTL für unendliche Transitionssysteme.** Wie bereits beschrieben, werden beim klassischen Model Checking die einzelnen Systemzustände durch die Wahrheitswerte *endlich* vieler Propositionen (wie $g$ und $w$ in Abb. 3) charakterisiert. In unendlichen Transitionssystemen ist die Menge der Eigenschaften, die in einem einzelnen Systemzustand gelten können, in der Regel unbeschränkt – z. B. könnte für eine Variable $x$ eine der Aussagen




$$\mathsf{G}\,[\exists z : (\mathit{bezahlt}(x, z) \wedge \mathit{preis}(x, z))\ \mathsf{B}\ \mathit{ausgeliefert}(x)]$$

***Abb. 4** Eine mögliche Entwicklung der Datenbank eines Online-Shops, und eine Formel, deren Gültigkeit verifiziert werden soll. Die Aussage der Formel ist, dass es immer (*G*(lobal)) gelten muss, dass der Preis $z$ der Ware $x$ bezahlt wurde, bevor (*B*) die Ware ausgeliefert wird. Dies gilt auch für die gegebene Folge von Datenbankzuständen*

$x = 0$, $x = 1$, $x = 2$, … gelten. Es liegt daher nahe, die beim klassischen Model Checking benutzten Logiken um die Möglichkeit zu erweitern, kompliziertere Aussagen über einzelne Systemzustände zu formulieren. Über einen möglichen Ansatz für eine Erweiterung von LTL in diesem Sinne wird in einer sehr lesenswerten Arbeit von Victor Vianu berichtet [26]. Wir betrachten diesen Ansatz anhand eines kleinen Beispiels.

*Ein Onlineshop.* Stellen wir uns vor, wir wollen einen Onlineshop eröffnen, der im Internet Rechner und Zubehör anbietet. Das System für unseren Onlineshop baut auf einer Datenbank auf und kann sogar als Schnittstelle zur Datenbank gesehen werden. Die Datenbank ist das Herzstück des Systems: Jede Anfrage an unsere Webseite wird Abfragen und Updates für die Datenbank zur Folge haben. Wir entwerfen also ein *datenbankbasiertes System*.

*Modellierung durch Transitionssysteme und Logiken.* Beim Entwurf eines Transitionssystems, das den Onlineshop modelliert, stellen wir fest, dass die interessanteste Eigenschaft eines aktuellen Zustands des Systems der aktuelle Zustand der Datenbank ist. Zur Spezifikation von erwünschten oder unerwünschten Systemeigenschaften wollen wir daher eine Logik verwenden, die über Datenbankzustände reden kann.

Welche Logik wir genau wählen, hängt davon ab, welche Datenbank wir nutzen. Wenn unsere Datenbank eine traditionelle relationale Datenbank ist, können wir die Prädikatenlogik verwenden. Wenn sie eine XML-basierte Datenbank ist, wollen wir vielleicht eine Variante der im nächsten Abschnitt erwähnten, für XML geeigneten Logiken nutzen. Auf jeden Fall wollen wir die Aussagensymbole (Propositionen) in LTL-Formeln durch Formeln der von uns gewählten Zustandslogik ersetzen (Abb. 4 zeigt ein Beispiel für eine entsprechende Formel). In jedem Zustand des Transitionssystems können wir dann diese Formeln über der aktuellen Datenbankinstanz auswerten, um herauszufinden, welche wahr und welche falsch sind.

*Herausforderungen.* Jetzt sieht es so aus, als ob wir einfach die klassischen Model-Checking-Techniken verwenden könnten. Aber es gibt immer noch ein paar Hindernisse: Das Transitionssystem ist nicht mehr endlich. Dies bedeutet, dass unendliche Durchläufe nicht unbedingt Zustände wiederholt besuchen müssen. Wir müssen außerdem damit rechnen, dass es eine unendlichen Menge von Anfangszuständen geben kann, da wir nicht wissen können, wie die Datenbank am Anfang eines Durchlaufes aussieht. Diese Herausforderungen zu bewältigen, ist derzeit Gegenstand der Forschung im Bereich des *Infinite State Model Checking*. Es werden z. B. passende Zustandslogiken gesucht und geeignete Einschränkungen an das Design des Systems gemacht, sodass die wichtigsten Eigenschaften effizient überprüft werden können.

## XML

Die eXtensible Markup Language (XML) hat sich im vergangenen Jahrzehnt als Standarddatenformat für den Datenaustausch im Internet etabliert. Ihre Stärke liegt in ihrer Flexibilität: XML ist sehr ähnlich zu HTML, mit dem entscheidenden Unterschied, dass in XML die *Tags* selbst gewählt werden können. XML-Dokumente haben also eine hierarchische Struktur, die in der Regel auf zwei Arten dargestellt wird: als Sequenz von Tags und Text (durch einen sogenannten *SAX-Datenstrom*) oder als Baum (in der sogenannten *DOM-Repräsentation*). Ein Beispiel eines XML-Dokuments, dargestellt

```
<Dokument>
    <Lager>
        <Laptop>
            <Marke>Lenovo</Marke>
            <Fabrikat>ThinkPad X201</Fabrikat>
            <Gewicht>1,54 kg</Gewicht>
        </Laptop>
        <Laptop>
            <Marke>Apple</Marke>
            <Fabrikat> MacBook Air </Fabrikat>
            <Gewicht>1,36 kg</Gewicht>
        </Laptop>
        ...
    </Lager>
</Dokument>
```

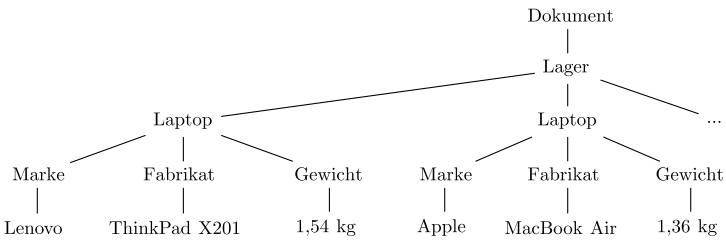



***Abb. 5 Ein XML-Dokument als Datenstrom (oben) und als Baum (unten)***

als Datenstrom und als Baum, findet sich in Abb. 5.

**Anfragesprachen und Schemasprachen.** Zur automatischen Verarbeitung von Dokumenten sind XML-Schemasprachen und XML-Anfragesprachen essenziell. Anfragesprachen (wie XPath, XQuery und XSLT) dienen dazu, Informationen aus XML-Dokumenten zu extrahieren – ähnlich wie SQL in relationalen Datenbanksystemen. Schemasprachen (z. B. XML Schema, DTDs oder RelaxNG) werden genutzt, um zu spezifizieren, wie Dokumente für einen gewissen Anwendungsbereich auszusehen haben, d. h. beispielsweise, welche hierarchische Struktur sie haben sollen. Dass ein Dokument einem gegebenen Schema entspricht, ist eine essenzielle Voraussetzung für Programme, die XML-Daten verarbeiten. Den Prozess des Überprüfens, ob ein gegebenes XML-Dokument einem bestimmten Schema entspricht, nennt man *Validierung*.

Wir betrachten diese Sprachen wieder als Logiken, wobei es bei Anfragesprachen nicht nur um die Spezifikation von Eigenschaften eines Dokuments, sondern auch um die Rückgabe eines Ergebnisses geht.

**Verbindung zu Automaten und klassischen Logiken.** Das Konzept der regulären Sprachen lässt sich nicht nur, wie oben gesehen, auf unendlich lange Strings übertragen, sondern auch auf Bäume, deren Knoten mit Symbolen aus einem endlichen Alphabet markiert sind. Reguläre Baumsprachen spielen eine wichtige Rolle bei der Untersuchung und Verwendung von XML-Schemabeschreibungen. Ein in einer der oben genannten Sprachen formuliertes Schema entspricht im Prinzip einem Baumautomaten, d. h. einer Variante der herkömmlichen endlichen Automaten, die als Eingabe Bäume anstelle von Strings verarbeiten. Und die regulären Baumsprachen sind gerade die Mengen von Bäumen, die sich durch Ausdrücke der *monadischen Logik zweiter Stufe* (MSO) beschreiben lassen (siehe z. B. [23]). Damit übersetzen sich viele Fragen über XML-Schemasprachen direkt in Fragen über Logiken und Automaten.

Aber auch die Standard-XML-Anfragesprachen haben enge Verbindungen zur Logik. Der Kern von XQuery zur Formulierung von nichtrekursiven Top-down-Anfragen entspricht der klassischen Prädikatenlogik [2]. XPath 1.0 entspricht im Wesentlichen der Einschränkung der Prädikatenlogik auf Ausdrücke, die nur zwei Variablen verwenden [19]; XPath 2.0 wurde bewusst so erweitert, dass es die Mächtigkeit der Prädikatenlogik umfasst. Des Weiteren besteht ein starker Zusammenhang zwischen XSLT und MSO [4].

Zur Bearbeitung von XML-bezogenen aus der Praxis motivierten Fragestellungen wurden in den

vergangenen Jahren in der Logik und der Automatentheorie neue Modelle entwickelt und erforscht, die speziell auf XML zugeschnitten sind. Einige der Fragestellungen, Ergebnisse und Forschungsrichtungen wollen wir im Folgenden etwas näher beleuchten.

**Effiziente Anfrageauswertung.** In der Untersuchung von Gottlob et al. [12] wurde experimentell bestätigt, dass alle damals verfügbaren XPath-Engines im „worst-case“ exponentielle Zeit zur Auswertung von XPath-Anfragen benötigten. Schon sehr kleine (wenn auch für den Zweck mutwillig konstruierte) Anfragen ließen sich nicht zu Ende führen. Die Autoren von [12] entwickelten ein einfaches, auf dynamischer Programmierung basierendes Verfahren, das in der Lage ist, (fast) die ganze Sprache XPath 1.0 effizient (d. h., in polynomieller Zeit) auszuwerten. In den vergangenen Jahren wurden automatenbasierte Methoden entwickelt, die dies noch verbessern und substanzielle Fragmente von XPath sogar in Linearzeit auswerten können [6, 20]. Dazu werden XPath-Anfragen in Baumautomaten umgewandelt, die dann mit ausgefeilten Methoden in linearer Zeit ausgewertet werden. Dieser Zusammenhang wird in Abb. 6 illustriert.

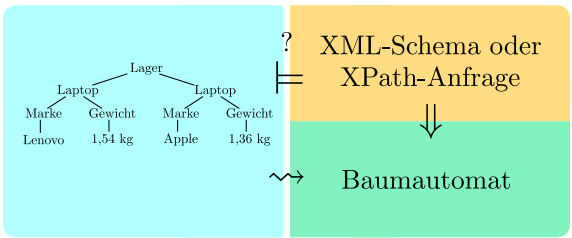



***Abb. 6 Zur Schemavalidierung und zur Anfrageauswertung werden Schemabeschreibungen und Anfragen in Baumautomaten überführt***

*Anfragen an Datenströme.* Wie lässt sich eine XPath-Anfrage auswerten, wenn die XML-Daten *nicht* in der DOM-Repräsentation als Baum, sondern nur als SAX-Datenstrom verfügbar sind? In einem SAX-Datenstrom wird das Dokument so gelesen, wie es in Abb. 5 (oben) dargestellt ist, und zwar Schritt für Schritt, von vorne nach hinten, und in nur *einem* Durchlauf. Da XPath-Engines für SAX-Datenströme oft viele Anfragen gleichzeitig evaluieren und einen möglichst großen Durchsatz erzielen müssen, ist es wichtig, dass sie mit wenig Speicherplatz auskommen. Welche XPath-Anfragen über einem SAX-Datenstrom mit konstantem Speicherplatz evaluiert werden können (und ähnliche Fragen) wurden beispielsweise in den Arbeiten [13, 14] untersucht.

*Statische Analyse von XPath-Ausdrücken.* Bei Anwendungen, die mit vielen XPath-Anfragen gleichzeitig umgehen, kann es sehr nützlich sein, das Verhältnis der Anfragen zueinander zu kennen. Wenn beispielsweise eine Anfrage $p$ für alle Dokumente den Wert TRUE liefert, für die auch Anfrage $q$ dies tut, ist eine Auswertung von $p$ für ein Dokument, für das $q$ TRUE geantwortet hat, nicht mehr nötig. Bei Anfragen mit Rückgabeergebnissen in einem Client/Server-System lässt sich durch solche Analysen beispielsweise der Kommunikationsaufwand verringern. Dies ist ein Beispiel für eine Anwendung des *Teilmengenproblems* (engl.: query containment problem). Das andere grundlegende Problem, das bei der statischen Analyse häufig gelöst werden muss, ist das in der Einleitung bereits erwähnte *Erfüllbarkeitsproblem*. Dabei geht es darum, für eine gegebene XPath-Anfrage auszuschließen, dass sie, unabhängig vom gegebenen XML-Dokument, immer das leere Ergebnis zurückliefert. Anfragen, die diesen Test nicht bestehen (aber möglicherweise kompliziert sind, weil sie automatisch erzeugt wurden), brauchen niemals ausgewertet zu werden.

Für die statische Analyse von XPath-Ausdrücken, Schemata etc. werden häufig automatenbasierte Methoden verwendet [22] – aufgrund des effizienten *Leerheitstests* und der Kombinierbarkeit durch Produkt- und Potenzmengenkonstruktion sind Automaten für solche Aufgaben nämlich hervorragend geeignet.

**Exkurs: Web-Wrapping zur Datenextraktion im Internet.** Um den Nutzen des Web-Wrappings zu illustrieren, betrachten wir ein Beispiel aus der Sicht des Betreibers eines Onlineshops. Um seine Angebote und Werbeaktionen zu optimieren, will der Betreiber täglich einen Überblick darüber bekommen, zu welchen Preisen seine Konkurrenten Produkte anbieten, die auch er in seinem Sortiment hat – etwa Laptops, Monitore und PCs der Marken Lenovo, Dell und Apple.

Die Informationen dazu kann er auf den Webseiten der Konkurrenten nachlesen. Das will er

natürlich nicht jeden Tag wieder von Hand tun; die Informationen sollen täglich automatisch zusammengestellt werden. Es gibt Systeme, die genau dies leisten, z. B. Lixto [11, 16].

*Die Grundidee:* Der Lixto-Nutzer meldet die Webseiten der Konkurrenten im System an und markiert für jede dieser Webseiten in einer grafischen Benutzungsoberfläche das, was für ihn von Interesse ist. In unserem Onlineshop-Beispiel sind das die Preise von Laptops, Monitoren und PCs der Marken Lenovo, Dell und Apple. Das Lixto-System generiert daraus automatisch Anfragen, die täglich gestartet werden, und es fasst die gefundenen Informationen in einer XML-Datei zusammen.

*Was steckt dahinter?* Indem der Lixto-Nutzer in der grafischen Oberfläche die für ihn interessanten Daten markiert, spezifiziert er eine Anfrage. Das System übersetzt dies in eine Anfrage, die in einer formalen Sprache formuliert ist, die Elog („Extraction by Datalog“) genannt wird [10, 11]. Die Sprache Elog wurde speziell für das Web-Wrapping entwickelt. Sie basiert auf der Logik *Monadisches Datalog*, einer Unterklasse der Sprache *Datalog*, die auch in vielen anderen Bereichen der Informatik genutzt wird.

Zur Auswertung von Elog-Anfragen werden Methoden zur Auswertung vom Monadischen Datalog genutzt. Es ist bekannt, dass Monadische-Datalog-Anfragen auf Bäumen (z. B. HTML- oder XML-Dokumenten) sehr effizient, in Zeit $O(a \cdot b)$ ausgewertet werden können, wobei $a$ und $b$ die Größe der jeweiligen Anfrage und des Baums bezeichnen [10]. Andererseits konnte unter Verwendung von Methoden der Automatentheorie nachgewiesen werden, dass Monadisches Datalog auf Bäumen genau dieselbe, große Ausdrucksstärke besitzt wie die Logik MSO [10].

**XML-Schemasprachen.** Der bereits erwähnte enge Zusammenhang zwischen XML-Schemasprachen und regulären Baumsprachen hat zu einer breiten Anwendung autmatentheoretischer Methoden in diesem Bereich geführt. Die automatentheoretische Sichtweise hat zu einem erheblich verbesserten Verständnis der verschiedenen Ad-hoc-Einschränkungen des W3C in den Standardschemasprachen geführt [8, 18].

Nicht zuletzt wurde in den vegangenen Jahren auch eine äußerst fruchtbare Verbindung zum maschinellen Lernen ausgenutzt: Viele im Web verfügbaren XML-Dokumente besitzen keine zugehörige Schemabeschreibung; viele Schemabeschreibungen sind syntaktisch nicht korrekt. Da für viele Anwendungen gute Schemainformation dringend benötigt werden, ist es nahe liegend, Schemabeschreibungen automatisch aus gegebenen XML-Dateien herzuleiten. Die in diesem Kontext entwickelten Schemainferenzmethoden bauen auf klassischen Techniken zum Lernen von regulären Sprachen auf [5].

## Von der Theorie zur Praxis und wieder zurück

Die in diesem Artikel vorgestellten Beispiele illustrieren das Zusammenspiel von Logik und Automaten sowie den überaus fruchtbaren Bezug zwischen der Forschung in Logik und Automatentheorie auf der einen und Anwendungen in der Informatik auf der anderen Seite. Diese Bezüge wirken natürlich auch auf die Entwicklung der Theorie zurück. Forschungsgegenstände, die bereits in den 1960er- bis 1980er-Jahren untersucht wurden, haben dadurch eine neue Aktualität gewonnen. Als Beispiele seien die Baumautomatenminimierung [17], die Theorie der Read/Write Streams und der mpms-Automaten [21] sowie die Entwicklung von Automaten für Strings und Bäume mit Datenwerten [7] genannt.

Manchmal stellt sich dabei heraus, dass trotz der Reife der betreffenden Gebiete noch längst nicht alle „einfachen“ Fragen geklärt sind. In einer Arbeit von Gelade und Neven [9] wurde z. B. kürzlich untersucht, wie groß ein kleinster regulärer Ausdruck $s$ für die Komplementsprache zu einem gegebenen regulären Ausdruck $r$ im schlimmsten Fall werden kann. Die Frage ergab sich im Zusammenhang der Beschreibung aller Dokumente, die ein gegebenes Schema *nicht* erfüllen. Erstaunliches Ergebnis: Es kann sein, dass $s$ mindestens etwa $2^{2^{|r|}}$ Zeichen lang sein muss, wobei $|r|$ die Länge des Audrucks $r$ angibt.

Dass Logik und Automaten für die Informatik eine große Rolle spielen, ist natürlich keine neue Entwicklung. Einige Highlights der vergangenen 80 Jahre sind in Abb. 7 zusammengestellt. Insgesamt lässt sich mit Moshe Vardi konstatieren: „*Logic and Automata: a Match Made in Heaven.*“[5]

[5] Auf diesen Titel eines eingeladenen Vortrags von Moshe Vardi bei der Konferenz ICALP 2003 [24] spielt natürlich auch der Titel unseres Artikels an.

**1930er-Jahre:** Alan M. Turing führt ein mathematisches Automatenmodell ein, das heute unter dem Namen *Turingmaschine* bekannt ist. Indem er das Verhalten einer solchen Maschine durch Formeln der Logik erster Stufe (FO) beschreibt, gelingt es ihm nachzuweisen, dass es keinen Algorithmus geben kann, der bei Eingabe einer FO-Formel entscheidet, ob diese Formel in der Standard-Arithmetik gilt.

**1940er- und 50er-Jahre:** Das heute unter dem Namen *endliche Automaten* bekannte Modell wird (unabhängig voneinander von verschiedenen Forschern) eingeführt – u. a. motiviert durch Fragestellungen beim Entwurf von logischen Schaltkreisen und zur Modellierung menschlicher neuronaler Aktivitäten. Stephen C. Kleene weist die Äquivalenz von endlichen Automaten und regulären Ausdrücken nach.

**1960er-Jahre:** Ken Thompson entwickelt einen effizienten Algorithmus zur Suche von durch reguläre Ausdrücke definierten Zeichenketten in Dateien. Varianten dieses Algorithmus werden heute von dem Unix-Kommando `grep` sowie von vielen Texteditoren genutzt.

Reguläre Ausdrücke und endliche Automaten werden von Compilern zur lexikalischen Analyse genutzt.

Büchi, Elgot, Trakhtenbrot, Rabin, Doner, Thatcher und Wright weisen nach, dass endliche Automaten auf Worten und Bäumen genau diejenigen Sprachen erkennen können, die durch Sätze der Monadischen Logik zweiter Stufe (MSO) beschrieben werden können. Dies liefert beispielsweise Verfahren zur automatischen Synthese von Schaltkreisen, deren Verhalten durch bestimmte logische Formeln spezifiziert wird.

**1970er-Jahre:** Edgar F. Codd führt das *relationale Modell* ein, das die Grundlage für moderne relationale Datenbanksysteme bildet. Die heute als Industriestandard verwendete Datenbankanfragesprache SQL beruht auf dem sogenannten Tupelkalkül, der eine Ausprägung der Logik erster Stufe ist.

**1980er-Jahre:** Vardi, Wolper und Sistla prägen den Begriff der *automatentheoretischen Methode* zur automatischen Verifikation von Programmen. Pnueli, Clarke, Emerson, Quielle und Sifakis zeigen auf, dass die Nutzung von temporalen Logiken und Fixpunktlogiken zur formalen Spezifikation von Systemen zu effizienten Methoden für die automatische Übersetzung von Spezifikationen in Automaten führt.

**1990er-Jahre:** Unter dem Stichwort *Model Checking* werden mächtige Verfahren zur Verifikation von Systemen mit endlich vielen Zuständen entwickelt, die auf temporalen Logiken und der automatentheoretischen Methode basieren.

**2000er-Jahre:** PSL wird Industriestandard. Logik und Automatentheorie nimmt Einfluss auf den Entwurf von XML-Anfrage- und Schemasprachen und die Implementierung von Auswertungs-Algorithmen.

***Abb. 7 Zeitlicher Überblick***

## Danksagung

Wir bedanken uns bei Ahmet Kara, Matthias Niewerth, und Thomas Zeume für sehr hilfreiche Kommentare.

## Literatur

1. Baier C, Katoen J-P (2008) Principles of Model Checking. The MIT Press, Cambridge, MA
2. Benedikt M, Koch C (2009) From XQuery to relational logics. ACM T Database Syst 34(4) Article 25
3. Bérard B, Bidoit M, Finkel A, Laroussinie F, Petit A, Petrucci L, Schnoebelen P (2001) Systems and Software Verification. Model-Checking Techniques and Tools. Springer, Heidelberg
4. Bex G, Maneth S, Neven F (2002) A formal model for an expressive fragment of XSLT. Inform Syst 27(1):21–39
5. Bex GJ, Neven F, Schwentick T, Vansummeren S (2010) Inference of concise regular expressions and DTDs. ACM T Database Syst 35(2) Article 11
6. Bojanczyk M, Parys P (2008) XPath evaluation in linear time. In: International Symposium on Principles of Database Systems (PODS), pp 241–250
7. Bojanczyk M, Muscholl A, Schwentick T, Segoufin L (2009) Two-variable logic on data trees and XML reasoning. J ACM 56(3) Article 13
8. Brüggemann-Klein A, Wood D (1998) One-unambiguous regular languages. Inform Comput 142(2):182–206
9. Gelade W, Neven F (2008) Succinctness of the complement and intersection of regular expressions. In: Annual Symposium on Theoretical Aspects of Computer Science (STACS), pp 325–336
10. Gottlob G, Koch C (2004) Monadic datalog and the expressive power of languages for Web information extraction. J ACM 51(1):74–113
11. Gottlob G, Koch C, Baumgartner R, Herzog M, Flesca S (2004) The Lixto data extraction project – back and forth between theory and practice. In: International Symposium on Principles of Database Systems (PODS), pp 1–12
12. Gottlob G, Koch C, Pichler R (2005) Efficient algorithms for processing XPath queries. ACM T Database Syst 30(2):444–491
13. Green TJ, Gupta A, Miklau G, Onizuka M, Suciu D (2004) Processing XML streams with deterministic automata and stream indexes. ACM T Database Syst 29(4): 752–788
14. Grohe M, Koch C, Schweikardt N (2007) Tight lower bounds for query processing on streaming and external memory data. Theor Comput Sci 380(1–2):199–217
15. Grumberg O, Veith H (eds) (2008) 25 Years of Model Checking – History, Achievements, Perspectives, vol 5000 of Lecture Notes in Computer Science. Springer, Heidelberg
16. Lixto Software. URL: http://www.lixto.com/, letzter Zugriff 8.4.2010
17. Martens W, Niehren J (2007) On the minimization of XML Schemas and tree automata for unranked trees. J Comput Syst Sci 73(4):550–583
18. Martens W, Neven F, Schwentick T (2007) Simple off the shelf abstractions for XML Schema. Sigmod RECORD 36(3):15–22
19. Marx M, de Rijke M (2005) Semantic characterizations of navigational XPath. Sigmod RECORD 34(2):41–46
20. Parys P (2009) XPath evaluation in linear time with polynomial combined complexity. In: International Symposium on Principles of Database Systems (PODS), pp 55–64
21. Schweikardt N (2009) Machine models for query processing. Sigmod RECORD 38(2):18–28
22. Schwentick T (2004) XPath query containment. Sigmod RECORD 33(1):101–109
23. Thomas W (1997) Languages, automata, and logic. In: Rozenberg G, Salomaa A (eds) Handbook of Formal Languages, vol III. Springer, New York, pp 389–455
24. Vardi MY (2003) Logic and automata: A match made in heaven. In: International Colloquium on Automata, Languages and Programming (ICALP), pp 64–65
25. Vardi MY (2008) From Church and Prior to PSL. In: Grumberg O, Veith H (eds) 25 Years of Model Checking – History, Achievements, Perspectives, vol 5000 of Lecture Notes in Computer Science. Springer, Heidelberg, pp 150–171
26. Vianu V (2009) Automatic verification of database-driven systems: a new frontier. In: International Conference Database Theory (ICDT), pp 1–13