2

Spezifikationssprachen

2.1 Anforderungen

Gemäß dem vereinfachten Informationsfluss-Diagramm (s. Abb. 1.5) beschreibt dieses Kapitel die Anforderungen an Ansätze zur Spezifikation eingebetteter Systeme.

Es mag immer noch Fälle geben, in denen die Spezifikation eines eingebetteten Systems in einer natürlichen Sprache, wie z.B. Englisch, festgehalten wird. Dieser Ansatz ist aber vollkommen ungeeignet, denn ihm fehlen wichtige Anforderungen an Spezifikationssprachen: es muss möglich sein, eine Spezifikation auf Vollständigkeit sowie auf Widerspruchsfreiheit zu prüfen, außerdem möchte man aus der Spezifikation mit einer systematischen Vorgehensweise eine Implementierung des Systems herleiten können. Deshalb sollte die Spezifikation in Form einer maschinenlesbaren, formalen Sprache erfolgen. Spezifikationssprachen für eingebettete Systeme sollten in der Lage sein, die folgenden Sachverhalte darzustellen[1]:

- **Hierarchie:** Der Mensch ist im Allgemeinen nicht in der Lage, Systeme zu verstehen, die viele Objekte (wie z.B. Zustände, Komponenten) enthalten, die in komplexem Zusammenhang miteinander stehen. Um ein reales System zu beschreiben, benötigt man mehr Objekte als ein Mensch erfassen kann. Die Einführung von Hierarchie-Ebenen ist der einzige Weg, dieses Problem zu lösen. Hierarchie kann so eingesetzt werden, dass ein Mensch immer nur eine kleine Anzahl von Objekten auf einmal überschauen muss.

  Es gibt zwei Arten von Hierarchien:

  - **Verhaltens-Hierarchien:** Verhaltenshierarchien enthalten Objekte, die notwendig sind, um das Verhalten des Gesamtsystems zu beschrei-

[1] Zur Erstellung dieser Liste wurden Informationen aus den Büchern von Burns et al. [Burns und Wellings, 1990], Bergé et al. [Bergé et al., 1995] und Gajski et al. [Gajski et al., 1994] verwendet.

ben. Hierarchische Zustände und verschachtelte Prozeduren sind Beispiele solcher Hierarchien.

– **Strukturelle Hierarchien:** Strukturelle Hierarchien beschreiben, wie das Gesamtsystem aus einzelnen physikalischen Komponenten zusammengesetzt ist.

  Eingebettete Systeme können z.B. aus Prozessoren, Speichern, Aktuatoren und Sensoren bestehen. Prozessoren wiederum enthalten Register, Multiplexer und Addierwerke. Multiplexer bestehen aus Gattern.

- **Zeitverhalten:** Da das Einhalten von Zeitbedingungen eine grundlegende Anforderung an eingebettete Systeme ist, muss es möglich sein, diese Zeitbedingungen explizit in der Spezifikation zu erfassen.
- **Zustandsorientiertes Verhalten:** Es wurde bereits in Kapitel 1 erwähnt, dass Automaten eine gute Darstellungsmöglichkeit für reaktive Systeme sind. Aus diesem Grund sollte es einfach sein, zustandsorientiertes Verhalten, wie es endliche Automaten zeigen, zu beschreiben. Klassische Automatenmodelle sind allerdings nicht ausreichend, da sie weder Zeitbedingungen noch Hierarchie unterstützen.
- **Ereignisbehandlung:** Da eingebettete Systeme oft reaktive Systeme sind, müssen Mechanismen zur Beschreibung von Ereignissen existieren. Solche Ereignisse können externe (von der Umwelt erzeugte) oder interne (von Komponenten des Systems erzeugte) Ereignisse sein.
- **Keine Hindernisse bei der Erzeugung von effizienten Implementierungen:** Da eingebettete Systeme effizient sein müssen, sollte die Spezifikationssprache eine effiziente Realisierung des Systems nicht behindern oder unmöglich machen.

  Beispiel: Angenommen, eine Spezifikationstechnik setze die Existenz einer virtuellen Maschine oder eines umfangreichen Simulators voraus. Dann müssen diese selbst bei kleinen Spezifikationen implementiert werden, was den Entwurf insgesamt nicht mehr effizient werden lassen kann.
- **Unterstützung für den Entwurf verlässlicher Systeme:** Spezifikationstechniken sollten den Entwurf von verlässlichen Systemen unterstützen. Beispielsweise sollte die Spezifikationssprache eine eindeutige Semantik haben und den Einsatz formaler Verifikationstechniken erlauben. Außerdem sollte es möglich sein, Sicherheits- und Authentizitäts-Anforderungen zu beschreiben.
- **Ausnahmeorientiertes Verhalten:** In vielen praktischen Systemen treten Ausnahmen auf. Um verlässliche Systeme entwerfen zu können, muss die Behandlung solcher Ausnahmesituationen einfach zu beschreiben sein. Es ist nicht ausreichend, wenn man die Ausnahmebehandlung z.B. für jeden einzelnen Zustand angeben muss, wie das etwa bei klassischen Automatenmodellen der Fall ist. Beispiel: In Abb. 2.1 soll die Eingabe `k` das Auftreten einer Ausnahme darstellen.

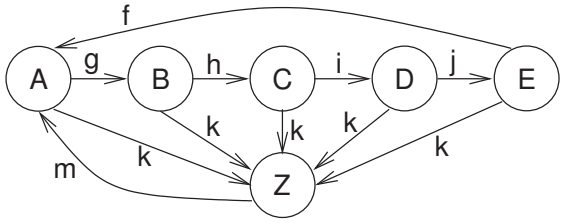

Abb. 2.1. Zustandsdiagramm mit Ausnahme k

Die Angabe einer solchen Ausnahme für jeden Zustand macht das Zustandsdiagramm sehr komplex und unübersichtlich, besonders bei großen Diagrammen mit vielen Transitionen. Wir werden später zeigen, wie man all diese Ausnahme-Transitionen durch eine einzige Transition ersetzen kann.

- **Nebenläufigkeit:** Viele praktisch relevante Systeme sind verteilte, nebenläufige Systeme. Aus diesem Grunde muss Nebenläufigkeit oder Parallelität einfach zu spezifizieren sein.
- **Synchronisation und Kommunikation:** Zeitgleich ablaufende Aktivitäten innerhalb eines Systems müssen miteinander kommunizieren und sich über die Verwendung von Ressourcen absprechen können. So ist es zum Beispiel häufig notwendig, gegenseitigen Ausschluss zu realisieren.
- **Programmiersprachenelemente:** Gewöhnliche Programmiersprachen haben sich als gängige Methode zur Beschreibung von Berechnungsvorschriften etabliert. Daher sollten Elemente von Programmiersprachen in der Spezifikation verwendet werden können. Klassische Zustandsdiagramme erfüllen diese Anforderung nicht.
- **Ausführbarkeit:** Eine Spezifikation ist nicht automatisch äquivalent zur ursprünglichen Idee des Designers. Das Ausführen der Spezifikation stellt eine Möglichkeit der Plausibilitätsprüfung für den Entwurf dar. Spezifikationen, die Programmiersprachenelemente verwenden, sind in diesem Kontext von Vorteil.
- **Unterstützung für den Entwurf großer Systeme:** Der Trend geht hin zu immer größeren und komplexeren Programmen, die auf eingebetteten Systemen ablaufen. In der Software-Technologie gibt es Mechanismen, um solche großen Systeme handhabbar zu machen, z.B. die Objektorientierung. Solche Mechanismen sollten auch in der Spezifikationsmethode Einsatz finden.
- **Unterstützung von spezifischen Anwendungsbereichen:** Es wäre natürlich schön, wenn man ein und dieselbe Spezifikationstechnik für alle möglichen eingebetteten Systeme verwenden könnte, da dies den Aufwand für die Entwicklung von Techniken minimieren würde. Allerdings ist die mögliche Bandbreite von Anwendungsgebieten so groß, dass man kaum hoffen kann, dass eine einzige Sprache alle Belange aller Anwen-

dungsbereiche gleichermaßen gut abdecken kann. Beispielsweise können kontrollflussdominierte, datenflussdominierte, zentralisierte oder verteilte Anwendungsgebiete von spezifischer *Tool*-Unterstützung für den jeweiligen Bereich profitieren.

- **Lesbarkeit:** Selbstverständlich muss eine Spezifikation von Menschen gelesen werden können. Nach Möglichkeit sollte sie auch maschinenlesbar sein, damit sie von einem Rechner verarbeitet werden kann.
- **Portierbarkeit und Flexibilität:** Spezifikationen sollten unabhängig von der für die Implementierung verwendeten spezifischen Hardware-Plattform sein, so dass man sie leicht für eine Auswahl von Zielplattformen einsetzen kann. Sie sollten so flexibel sein, dass eine kleine Änderung am Design auch nur eine kleine Änderung in der Spezifikation erfordert.
- **Terminierung:** Es sollte möglich sein, anhand der Spezifikation Prozesse zu identifizieren, die terminieren. Auf diese Weise wird erreicht, dass Prozessen, die keine Aufgaben mehr erfüllen, auch ihre Ressourcen entzogen werden können und dass man sich bei der Validierung mit diesen Prozessen nicht mehr beschäftigen muss.
- **Unterstützung für Nicht-Standard-Ein-/Ausgabe-Geräte:** Viele eingebettete Systeme verwenden andere Ein- und Ausgabegeräte als die vom PC her bekannten Tastaturen und Mäuse. Es sollte möglich sein, die Eigenschaften solcher Geräte in einfacher Weise zu spezifizieren.
- **Nicht-funktionale Eigenschaften:** Reale Systeme besitzen eine Reihe nicht-funktionaler Eigenschaften, wie etwa Ausfallsicherheit, Größe, Erweiterbarkeit, Lebenserwartung, Energieverbrauch, Gewicht, Recycling-Fähigkeit, Benutzerfreundlichkeit, elektromagnetische Verträglichkeit und so weiter. Es gibt keine Hoffnung, dass man all diese Eigenschaften in irgendeiner Weise formal definieren könnte.
- **Angemessenes Berechnungsmodell:** Zur Beschreibung von Berechnungen benötigt man ein Berechnungsmodell. Solche Modelle werden im nächsten Abschnitt behandelt.

An dieser Liste von Anforderungen kann man erkennen, dass es wohl nie eine einzige formale Sprache geben wird, die alle diese Anforderungen erfüllt. In der Praxis muss man daher in der Regel mit Kompromissen leben. Die Auswahl der Spezifikationssprache, die für ein bestimmtes Projekt verwendet wird, hängt hauptsächlich vom Anwendungsbereich und von der Umgebung, in der das System entwickelt werden soll, ab. Wir stellen im Folgenden einige Sprachen vor, die für den Entwurf eines praxisnahen Systems in Frage kommen.

2.2 Berechnungsmodelle

Die Anwendungen der Informationstechnologie waren bislang sehr stark an das Von-Neumann-Paradigma des sequentiellen Rechnens angelehnt. Dieses Paradigma ist für eingebettete Systeme, insbesondere für Realzeitsysteme, ungeeignet, da es beim Von-Neumann-Modell keine Möglichkeit gibt, Zeit auszudrücken. Andere Berechnungsmodelle sind für diesen Bereich eher geeignet.

Berechnungsmodelle definieren (in Anlehnung an [Lee, 1999]):

- **Komponenten** und die Organisation des Ablaufs von Berechnungen in diesen Komponenten: Prozeduren, Prozesse, Funktionen, endliche Automaten sind mögliche Komponenten.
- **Kommunikations-Protokolle**: Diese Protokolle beschreiben die Kommunikationsmöglichkeiten zwischen den Komponenten. Asynchroner Nachrichtenaustausch und *Rendez-Vous*-basierte Kommunikation sind Beispiele für solche Protokolle.

Beispiele für Berechnungsmodelle (siehe Lee [Lee, 1999], Janka [Janka, 2002] und Jantsch [Jantsch, 2003], die auch Nebenläufigkeit ausdrücken, können nach dem Modell der Kommunikation und dem Modell der Berechnungen in den Komponenten klassifiziert werden:

- **Modelle der Kommunikation von Komponenten**:
  - **Gemeinsamer Speicher** (*shared memory*)

    Bei diesem Modell greifen kommunizierende Komponenten auf gemeinsamen Speicher zu. Eine akzeptable Geschwindigkeit ergibt sich dabei nur für Komponenten, die sich in räumlicher Nähe zum Speicher befinden. Für verteilte Systeme ist dieses Modell also nicht geeignet.
  - **Nachrichtenaustausch** (*message passing*)

    Beim Nachrichtenaustausch kommunizieren Prozesse durch das Versenden von Nachrichten miteinander. Der Versand erfolgt über **Kanäle**. Wir unterscheiden zwischen asynchronem Nachrichtenaustausch und synchronem Nachrichtenaustausch.

    Beim **asynchronen Nachrichtenaustausch** muss der Absender nicht warten, bis der Empfänger bereit ist, die Nachricht zu empfangen. Er kann vielmehr in seinen Berechnungen fortfahren. Im Alltag entspricht das etwa dem Abschicken eines Briefes. In diesem Fall erfolgt der Versand über Kanäle, die in der Lage sind, Nachrichten zu puffern. Die meist notwendige Zwischenspeicherung von Nachrichten ist problematisch, da es zu Überläufen der Nachrichtenpuffer kommen kann.

    Bei **synchronem Nachrichtenaustausch** müssen Sender und Empfänger aufeinander warten (man spricht deshalb auch von der *rendez-*

vous-Technik). Kanäle müssen hierbei nicht puffern. Dafür ergibt sich dann allerdings häufig eine Leistungseinbuße in den Komponenten.

- **Modelle der Berechnungen in den Komponenten**
  - **Von Neumann-Modell**

    Als Modell der Berechnung in den Komponenten können wir z.B. das von-Neumann-Modell einer sequentiellen Abarbeitung eines Befehlsstroms benutzen. Hierbei wird die Existenz von abzuarbeitenden Maschinenbefehlen vorausgesetzt.
  - **Diskretes Ereignismodell**

    In diesem Modell tragen alle Ereignisse einen total geordneten Zeitstempel, der die Zeit angibt, zu der das Ereignis stattfindet. Simulatoren für diskrete Ereignismodelle verfügen über eine nach der Zeit sortierte globale Ereignis-Warteschlange. Beispiele sind u.a. VHDL (s. Seite 64) und Verilog (s. Seite 81).
  - **Endliche Automaten**

    Klassische endliche Automaten bilden ein weiteres Modell der Berechnung in Komponenten. Ihr Verhalten wird durch die üblichen Übergangs- und Ausgabefunktionen beschrieben.
- **Kombinierte Modelle**

  Manche Modelle verbinden ein bestimmtes Modell der Berechnung in den Komponenten mit einem Modell der Kommunikation. So kombiniert SDL (s. Seite 32) beispielsweise das Modell eines endlichen Automaten für die Berechnung in den Komponenten mit dem Modell des asynchronen Nachrichtenaustauschs für die Kommunikation. StateCharts dagegen kombiniert das Modell des endlichen Automaten für die einzelne Komponente mit dem Modell des gemeinsamen Speichers für die Kommunikation. CSP (s. Seite 60) und ADA (s. Seite 61) kombinieren das Modell der von-Neumann-Befehlsausführung mit dem synchronen Nachrichtenaustausch.

Verschiedene Anwendungen können die Verwendung unterschiedlicher Modelle notwendig machen. Während einige der verfügbaren Spezifikationssprachen nur eines der Modelle implementieren, erlauben andere eine Mischung verschiedener Modelle. Für die Mehrzahl der Modelle werden nachfolgend exemplarisch Beispielsprachen vorgestellt.

2.3 StateCharts

Die erste praktisch eingesetzte Sprache, die hier vorgestellt werden soll, ist StateCharts. StateCharts wurde 1987 von David Harel [Harel, 1987] vorgestellt und später detaillierter beschrieben [Drusinsky und Harel, 1989]. StateCharts beschreibt kommunizierende endliche Automaten, basierend auf dem

shared memory Kommunikations-Konzept. Den Namen hat Harel angeblich so gewählt, weil es „*die einzige unbenutzte Kombination von 'flow' oder 'state' mit 'diagram' oder 'chart'* “ war.

Im Abschnitt 2.1 wurde erwähnt, dass es oft notwendig ist, zustandsorientiertes Verhalten zu modellieren. Zustandsdiagramme sind dafür die klassische Methode. Abb. 2.2 (identisch mit Abb. 2.1) zeigt ein Beispiel für ein klassisches Zustandsdiagramm, das einen **endlichen Automaten** darstellt.

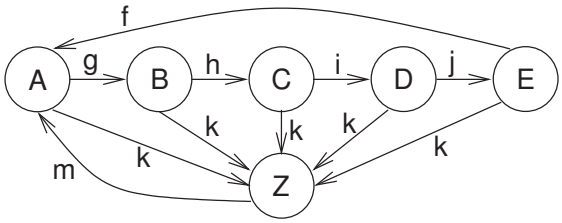

Abb. 2.2. Zustandsdiagramm

Kreise bezeichnen Zustände. **Deterministische** endliche Automaten, die wir hier betrachten wollen, können sich immer nur in einem ihrer Zustände befinden. Kanten stellen die Transitionen oder Übergänge zwischen den Zuständen dar. Die Kantenbeschriftung bezeichnet Ereignisse. Wenn eine Eingabe stattfindet, ändert der endliche Automat seinen Zustand der jeweiligen Kante entsprechend. Endliche Automaten können auch eine Ausgabe erzeugen (dies ist in Abb. 2.2 nicht gezeigt). Nähere Informationen zu klassischen endlichen Automaten findet man z.B. in [Kohavi, 1987].

2.3.1 Modellierung von Hierarchie

StateCharts beschreiben erweiterte endliche Automaten und sind daher gut geeignet, zustandsorientiertes Verhalten abzubilden. Die wichtigste Erweiterung gegenüber klassischen Automaten ist das Konzept der **Hierarchie**. Hierarchie wird mit Hilfe von **Superzuständen** (*superstates*) eingeführt.

Definitionen:

- Zustände, die andere Zustände enthalten, heißen **Superzustände**
- Zustände, die in anderen Zuständen enthalten sind, heißen **Unterzustände** der Superzustände

Abb. 2.3 zeigt ein StateCharts-Beispiel. Es ist eine hierarchische Version von Abb. 2.2.

Superzustand S beinhaltet die Zustände A, B, C, D und E. Angenommen, der Automat befindet sich in Zustand Z (wir bezeichnen Z in diesem Fall auch als **aktiven Zustand**). Wenn dann die Eingabe m am Automaten erfolgt, ist A

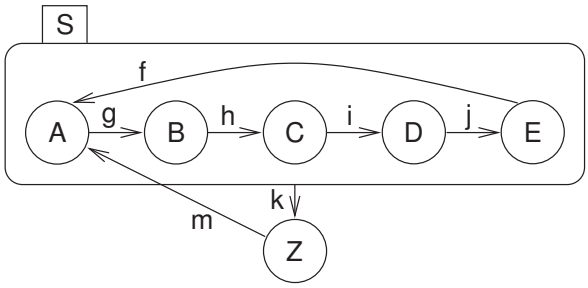

Abb. 2.3. Hierarchisches Zustandsdiagramm

der nächste Zustand. Wenn sich der Automat in Zustand S befindet und es gibt die Eingabe k, so ist Z der nächste Zustand, unabhängig davon, in welchem der Unterzustände A, B, C, D oder E sich der Automat tatsächlich befindet. In diesem Beispiel sind alle in S enthaltenen Zustände nicht-hierarchische Zustände. Im Allgemeinen könnten die Unterzustände von S selbst wieder Superzustände sein, die weitere Unterzustände enthalten.

Definitionen:

- Jeder Zustand, der nicht aus anderen Zuständen besteht, heißt **Basiszustand**.
- Ist t ein Unterzustand von s, so heißt s **Oberzustand** von t.

Der endliche Automat in Abb. 2.3 kann sich zu einem bestimmten Zeitpunkt nur in genau einem der Unterzustände des Superzustands S befinden. Solche Superzustände heißen **ODER-Superzustände**. Gemeint ist hier immer ein sich gegenseitig ausschließendes ODER, da sich der Automat nur in **einem** der Zustände A, B, C, D oder E befinden kann.

Definition: Superzustände S heißen **ODER-Superzustände**, wenn das System, das S enthält, sich zu jedem Zeitpunkt lediglich in einem einzigen Unterzustand von S befinden kann, solange es sich in S befindet.

In Abb. 2.3 könnte die Eingabe k einer Ausnahme entsprechen, wegen der Zustand S verlassen werden muss. Das Beispiel zeigt bereits, wie man solche Ausnahmebehandlungen durch Verwendung von Hierarchie kompakt darstellen kann.

StateCharts erlaubt es, Systeme hierarchisch zu beschreiben, indem eine Beschreibung des Systems Untersysteme enthält, die wiederum Beschreibungen von Unterzuständen enthalten können und so weiter. Das Gesamtsystem kann somit in Form eines **Baumes** dargestellt werden. Die **Wurzel** dieses Baumes entspricht dem gesamten System, und die inneren Knoten entsprechen hierarchischen Beschreibungen (die im Falle von StateCharts Superzustände heißen). Die **Blätter** der Hierarchie sind nicht-hierarchische Beschreibungen (die im Falle von StateCharts Basiszustände heißen).

Bisher haben wir explizite, direkte Kanten verwendet, die zu Basiszuständen führen, um den neuen Zustand zu bestimmen. Der Nachteil dieser Art der Modellierung liegt in der Tatsache, dass man die internen Strukturen der Superzustände nicht vor der Umgebung verstecken kann. In einem echten hierarchischen Modell sollte es möglich sein, die interne Struktur zu verstecken, so dass diese später beschrieben oder sogar geändert werden kann, ohne eine Auswirkung auf die Umgebung zu haben. Dies wird durch zusätzliche Mechanismen ermöglicht, die den neuen Zustand bestimmen.

Der erste zusätzliche Mechanismus ist der **Standardzustand** (*default state*). Er kann in Superzuständen verwendet werden, um anzuzeigen, welche der Unterzustände betreten werden, wenn der Superzustand betreten wird. In den Diagrammen wird der Standardzustand mit Hilfe einer Kante bezeichnet, die von einem kleinen ausgefüllten Kreis zum jeweiligen Zustand geht. Abb. 2.4 zeigt ein Zustandsdiagramm, das den Standardzustands-Mechanismus verwendet. Es ist äquivalent zum Diagramm in Abb. 2.3. Man beachte, dass der kleine ausgefüllte Kreis selber keinen Zustand darstellt.

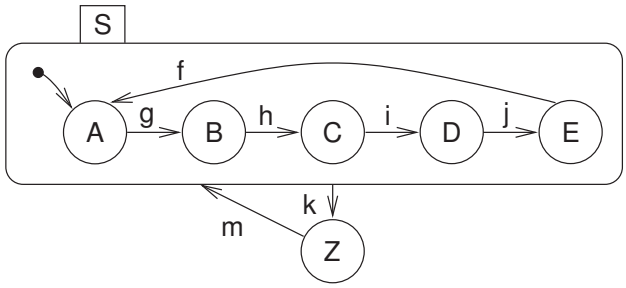

Abb. 2.4. Zustandsdiagramm mit Standardzustand

Ein weiterer Mechanismus, um den nächsten Zustand anzugeben, ist der sogenannte ***History*-Zustand**. Mit Hilfe dieses Konstrukts ist es möglich, in den letzten Unterzustand zurückzukehren, der aktiv war, bevor der Superzustand verlassen wurde. Der *History*-Mechanismus wird durch den Buchstaben H in einem Kreis dargestellt. Um den aktiven Zustand für den ersten Wechsel in einen Superzustand zu kennzeichnen, wird der *History*-Mechanismus oft mit einem Standardzustand kombiniert. Abb. 2.5 zeigt ein Beispiel.

Das Verhalten des endlichen Automaten hat sich nun verändert: Sei der Automat zunächst im Zustand Z, und es ereigne sich die Eingabe m. Dann ist A der nächste Zustand, wenn der Superzustand S zum ersten Mal betreten wird. Ansonsten wird der zuletzt aktive Unterzustand betreten. Dieser Mechanismus hat viele Anwendungen. Wenn z.B. die Eingabe k eine Ausnahme darstellt, könnte die Eingabe m verwendet werden, um in den Zustand vor der Ausnahme zurückzukehren. Die Zustände A, B, C, D und E könnten Zustand Z auch wie eine Prozedur aufrufen. Nachdem diese „Prozedur“ Z abgearbeitet ist, kehrt der Automat zum aufrufenden Zustand zurück.

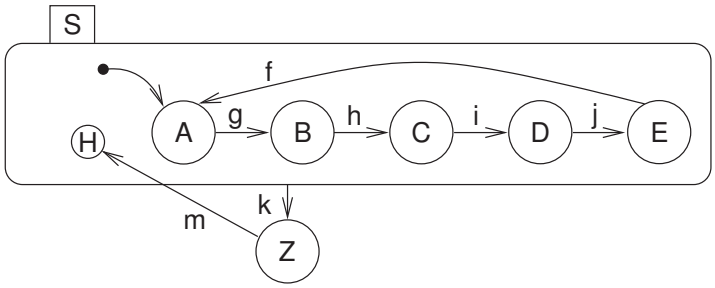

Abb. 2.5. Zustandsdiagramm mit *History*-Mechanismus und Standardzustand

Der Automat aus Abb. 2.5 kann auch wie in Abb. 2.6 dargestellt werden. In diesem Fall wurden die Darstellungen für den Standardzustand und den *History*-Mechanismus kombiniert.

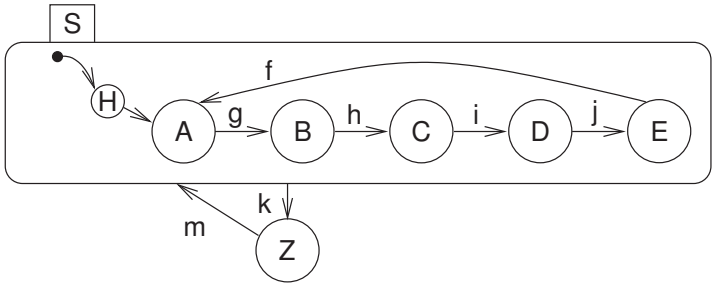

Abb. 2.6. Kombination der Symbole für *History*- und Standardzustand

Eine Spezifikationstechnik muss auch in der Lage sein, Nebenläufigkeit und Parallelität darzustellen. Zu diesem Zweck gibt es in StateCharts eine zweite Art von Superzuständen, die sogenannten **UND-Superzustände**.

Definition: Superzustände S heißen **UND-Superzustände**, wenn das System, das S enthält, sich in allen Unterzuständen von S gleichzeitig befindet, solange es sich in S befindet.

Das Diagramm für einen Anrufbeantworter in Abb. 2.7 enthält einen UND-Superzustand.

Ein Anrufbeantworter muss normalerweise zwei Aufgaben parallel ausführen: er wartet auf ankommende Anrufe und überprüft gleichzeitig, ob der Benutzer etwas auf den Eingabetasten eingegeben hat. In Abb. 2.7 heißen die entsprechenden Zustände Lwait und Kwait. Ankommende Anrufe werden im Zustand Lproc abgearbeitet, während Reaktionen auf die Eingabetasten im Zustand Kproc erzeugt werden. Der Ein-/Aus-Schalter wird fürs Erste so modelliert, dass die von ihm erzeugten Ereignisse (Einschalten und Ausschalten) separat dekodiert werden und somit nicht in den Zustand Kproc gewechselt wird. Wird der Anrufbeantworter ausgeschaltet, so werden die beiden im ein-Zustand enthaltenen Zustände verlassen. Sie werden erst wieder betreten, wenn der Anrufbeantworter wieder eingeschaltet wird. Zu diesem Zeitpunkt werden dann die

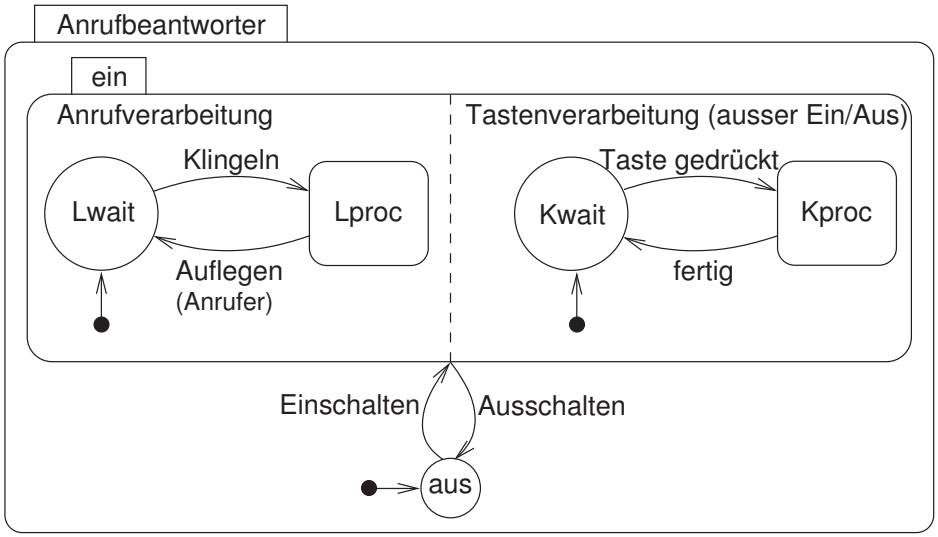

Abb. 2.7. Anrufbeantworter

Standardzustände Lwait und Kwait betreten. Solange die Maschine eingeschaltet ist, befindet sie sich immer in den beiden Unterzuständen des Zustands ein.

Für UND-Superzustände können die Unterzustände, die als Reaktion auf ein Ereignis betreten werden, unabhängig voneinander beschrieben werden. Jede beliebige Kombination von *History*-, Standardzuständen und expliziten Transitionen ist möglich. Für das Verständnis von StateCharts ist es wichtig zu begreifen, dass immer **alle** Unterzustände betreten werden, auch wenn es nur eine einzige explizite Transition zu einem der Unterzustände gibt. Analog dazu gilt, dass eine Transition aus einem UND-Superzustand heraus immer dazu führt, dass **alle** seine Unterzustände verlassen werden.

Als Beispiel modifizieren wir unseren Anrufbeantworter so, dass der Ein-/Aus-Schalter wie alle anderen Bedientasten im Zustand Kproc dekodiert und behandelt wird (s. Abb. 2.8).

Wenn der Anrufbeantworter ausgeschaltet wird, findet eine Transition in den aus-Zustand statt. Dieser Übergang führt dazu, dass auch der Zustand, der auf ankommende Anrufe wartet, verlassen wird. Das Wiedereinschalten der Maschine führt dazu, dass eben dieser Zustand auch wieder mit betreten wird.

Zusammenfassend können wir festhalten: **Zustände in StateCharts-Diagrammen sind entweder UND-Superzustände, ODER-Superzustände oder Basiszustände.**

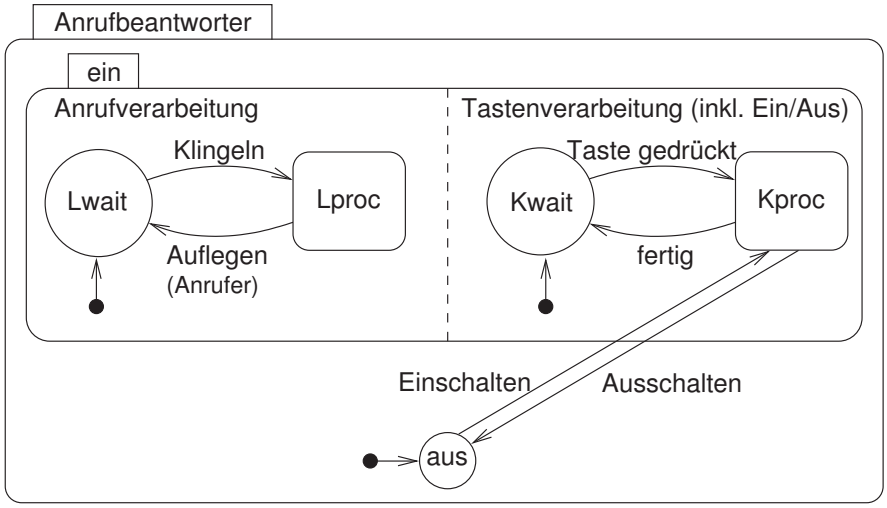

Abb. 2.8. Anrufbeantworter mit veränderter Ein-/Ausschalter-Modellierung

2.3.2 Zeitbedingungen

Da es notwendig ist, in eingebetteten Systemen Zeitbedingungen zu modellieren, bietet StateCharts die sogenannten *Timer* an. Zeitbedingungen werden durch das gezackte Symbol im linken Teil von Abb. 2.9 dargestellt.

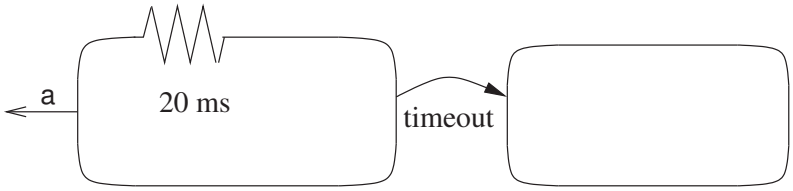

Abb. 2.9. Zeitbedingungen in StateCharts

Wenn das System für die im *Timer* angegebene Zeitdauer im *Timer*-Zustand war, wird ein *Time-Out* ausgelöst und das System verlässt diesen Zustand. *Timer* können auch hierarchisch verwendet werden.

Ein *Timer* könnte beispielsweise in einer tieferen Hierarchiestufe des Anrufbeantworters verwendet werden, um das Verhalten des Zustands Lproc zu beschreiben. Abb. 2.10 zeigt eine mögliche Beschreibung für diesen Zustand.

Da das Auflegen des Anrufers in Abb. 2.7 in Form einer Ausnahmebehandlung realisiert ist, wird der Zustand Lproc immer erst dann verlassen, wenn der Anrufer auflegt. Wenn allerdings der Angerufene auflegt, hat der Entwurf des Zustands Lproc einen Schönheitsfehler: wenn der Angerufene zuerst auflegt, ist das Telefon solange tot (und still), bis der Anrufer ebenfalls aufgelegt hat.

StateCharts beinhalten noch weitere Sprachelemente, die beispielsweise im Buch von Harel [Harel, 1987] zu finden sind. Zusammen mit Drusinsky gibt

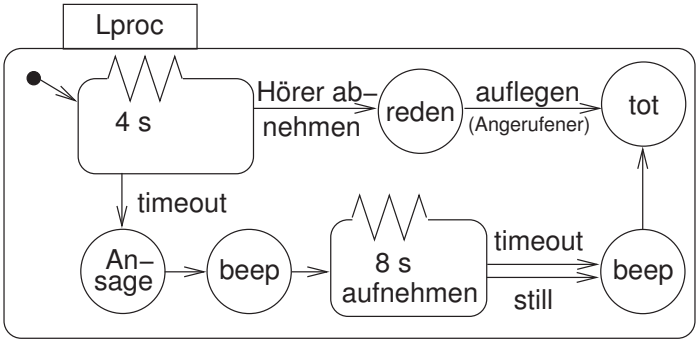

Abb. 2.10. Behandlung von eingehenden Anrufen in Lproc

Harel [Drusinsky und Harel, 1989] in einem Artikel eine genauere Beschreibung der Semantik von StateMate, einer StateCharts-Implementierung.

2.3.3 Kantenbeschriftungen und StateMate-Semantik

Die Ausgabe der erweiterten Automatenmodelle wurde bislang noch nicht betrachtet. Solche Ausgaben können mit Hilfe von Kantenbeschriftungen realisiert werden. Die allgemeine Form einer Kantenbeschriftung ist „Ereignis [Bedingung] / Reaktion“. Alle drei Bestandteile der Beschriftung sind optional. Die **Reaktion** beschreibt die Reaktion des Automaten auf den Zustandsübergang. Mögliche Reaktionen beinhalten das Erzeugen von Ereignissen oder die Zuweisung von Variablenwerten. Die Anteile Bedingung und Ereignis beschreiben zusammen die Eingaben an den Automaten. Die **Bedingung** beschreibt das Überprüfen von Werten von Variablen oder des Zustands des Gesamtsystems. Der **Ereignis**teil symbolisiert, auf welches Ereignis zu prüfen ist. Solche Ereignisse können entweder intern oder extern erzeugt werden. Interne Ereignisse werden als Ergebnis von Zustandsübergängen erzeugt und werden in der Reaktion des Übergangs beschrieben. Externe Ereignisse werden üblicherweise in der Systemumgebung beschrieben.

Beispiele:

- Einschalten / Ein:=1 (Test auf ein Ereignis und Wertzuweisung an eine Variable),
- [Ein=1] (Überprüfung eines Variablenwerts),
- Ausschalten [not in Lproc] / Ein:=0 (Ereignistest, Überprüfung eines Zustands, Variablenzuweisung. Die Zuweisung wird nur durchgeführt, wenn das Ereignis stattgefunden hat und die Bedingung erfüllt ist).

Die Menge aller Variablenwerte zusammen mit der Menge von erzeugten Ereignissen (und die aktuelle Zeit) ist definiert als der **Status**[2] eines StateCharts-

[2] Normalerweise würde man den Begriff „Zustand“ statt „Status“ verwenden, aber der Begriff „Zustand“ hat in StateCharts eine andere Bedeutung.