Zeitschrift: Bulletin des Schweizerischen Elektrotechnischen Vereins, des Verbandes Schweizerischer Elektrizitätsunternehmen = Bulletin de l'Association suisse des électriciens, de l'Association des entreprises électriques suisses

Herausgeber: Schweizerischer Elektrotechnischer Verein ; Verband Schweizerischer Elektrizitätsunternehmen

Band: 80 (1989)

Heft: 17

Artikel: JSD : eine wirkungsvolle Methode zur Entwicklung von Softwaresystemen

Autor: Renold, A.

DOI: https://doi.org/10.5169/seals-903712

Nutzungsbedingungen

Die ETH-Bibliothek ist die Anbieterin der digitalisierten Zeitschriften auf E-Periodica. Sie besitzt keine Urheberrechte an den Zeitschriften und ist nicht verantwortlich für deren Inhalte. Die Rechte liegen in der Regel bei den Herausgebern beziehungsweise den externen Rechteinhabern. Das Veröffentlichen von Bildern in Print- und Online-Publikationen sowie auf Social Media-Kanälen oder Webseiten ist nur mit vorheriger Genehmigung der Rechteinhaber erlaubt. Mehr erfahren

Conditions d'utilisation

L'ETH Library est le fournisseur des revues numérisées. Elle ne détient aucun droit d'auteur sur les revues et n'est pas responsable de leur contenu. En règle générale, les droits sont détenus par les éditeurs ou les détenteurs de droits externes. La reproduction d'images dans des publications imprimées ou en ligne ainsi que sur des canaux de médias sociaux ou des sites web n'est autorisée qu'avec l'accord préalable des détenteurs des droits. En savoir plus

Terms of use

The ETH Library is the provider of the digitised journals. It does not own any copyrights to the journals and is not responsible for their content. The rights usually lie with the publishers or the external rights holders. Publishing images in print and online publications, as well as on social media channels or websites, is only permitted with the prior consent of the rights holders. Find out more

Download PDF: 09.12.2025

ETH-Bibliothek Zürich, E-Periodica, https://www.e-periodica.ch

JSD – eine wirkungsvolle Methode zur Entwicklung von Softwaresystemen

A. Renold

Jackson System Development (JSD) basiert auf dem operationellen Ansatz der Softwareentwicklung und ist gleichermassen für kommerzielle und Echtzeitsysteme geeignet. Viele bekannte Konzepte aus anderen Methoden finden sich in JSD wieder; sie sind aber anders angeordnet, was den Entwickler zu tieferer Einsicht in das zu entwickelnde System führt.

La méthode Jackson System Development (JSD) se base sur la conception opérationnelle du développement de logiciel. Elle convient tant aux systèmes de gestion qu'aux applications en temps réel. On y retrouve plusieurs concepts connus provenant d'autres méthodes, mais ils sont arrangés différemment, ce qui permet de mieux comprendre le système à développer.

Adresse des Autors:
André Renold, dipl. El.-Ing. ETH,
M-Informatic AG, Grünaustrasse 23,
8953 Dietikon.

Jackson System Development (JSD) ist eine Methode zur Entwicklung von Softwaresystemen. Sie deckt den technischen Aspekt eines Softwareprojekts von den Anforderungen bis zur Wartung ab. JSD wurde von *Michael Jackson* (nicht dem Sänger!) in London entwickelt und 1981 erstmals veröffentlicht. Eine Reihe von nachfolgenden Publikationen [1 . . . 5] zeigt, dass sich die Methode anhand der Erfahrungen aus ungefähr 200 Projekten fortlaufend weiterentwickelt.

JSD ist aus Jackson Structured Programming (JSP) entstanden. JSP ist eine Programm-Entwurfsmethode, welche auch unter dem Namen Jackson Design Methodology (JDM) bekannt ist. JSP ist ein integrierender Bestandteil von JSD. JSD leitet den Entwickler an, wie er ein System aus einzelnen Prozessen zusammensetzen kann, wobei die einzelnen Prozesse mit JSP entwickelt werden. JSD und JSP basieren auf den gleichen theoretischen Konzepten und gehen nahtlos ineinander über.

Leider wird die Jackson-Methode oft mit kommerziellen Anwendungen und Cobol assoziiert. Sie ist aber genausogut für Echtzeitsysteme (Embedded Systems) geeignet. Es ist richtig, dass die Jackson-Methode im kommerziellen Bereich entstanden ist, sie wurde aber ebenso erfolgreich im technischen Bereich eingesetzt (z.B. [6]). JSD basiert wesentlich auf dem Hoareschen Ansatz asynchron miteinander kommunizierender Prozesse [7], der sich in der technischen Welt gut bewährt hat.

Verglichen mit anderen Methoden, wie z.B. Structured Analysis/Structured Design, Mascot, SADT, geht JSD von einem radikal anderen Denkansatz aus. Man hat oft das Gefühl, JSD zäume das Pferd am Schwanz auf, realisiert dann aber, dass genau diese unterschiedliche Sichtweise zu neuen Einsichten und tieferem Verständnis führt. Dadurch wird es einfacher, korrekte Systeme zu entwickeln, was auch den Zeitaufwand für das Testen und die Wartung reduziert. Natürlich ist es oft schwierig, sich eine neue Denkweise anzueignen. Dies mag auch einer der Gründe sein, warum sich JSD noch nicht auf breiterer Front durchgesetzt hat.

Im vorliegenden Beitrag sollen die Grundkonzepte und das Vorgehen von JSD vorgestellt und anhand von Vergleichen mit anderen Methoden und Konzepten verdeutlicht werden.

Methodenansätze

Der konventionelle Ansatz in der Softwareentwicklung trennt die Definition des äusseren Verhaltens eines Systems, das *Was* (Anforderungsspezifikation), von der Definition der inneren Mechanismen des Systems, das *Wie* (Entwurfsspezifikation). Das Grundkonzept ist dabei im wesentlichen eine hierarchische Top-Down-Aufgliederung von Black Boxes. Ziel dieser Aufgliederung ist, in der feinsten Aufteilung ausführbare Code-Module zu erzeugen. Die Aufgliederung ist darauf ausgerichtet, ein System zu erzeugen, das optimal auf die gewünschte Laufzeitumgebung ausgerichtet ist und darin effizient abläuft. Vertreter des konventionellen Ansatzes sind z.B. Structured Analysis/Structured Design (SASD) [7; 8] und Mascot.

Im Gegensatz zum konventionellen Ansatz geht der operationelle Ansatz [9] davon aus, dass die *operationellen* Belange von den *implementationsorientierten* Belangen getrennt werden. Dies wird mit einer Spezifikation erreicht, die nur problemrelevante Tatsachen enthält, also noch keine Implementation vorwegnimmt. Diese Spezi-

fikation ist so formal, dass sie mit einem geeigneten Interpreter direkt ausführbar ist. Sie kann dadurch als Prototyp dienen, der aber unter Umständen noch zu wenig effizient abläuft.

In einem zweiten Schritt wird die Implementation mit Hilfe von bedeutungserhaltenden Transformationen aus der Spezifikation abgeleitet. Die Transformationen werden dabei so ausgewählt, dass das implementierte System die verlangten Zeitanforderungen erfüllt, d.h. die Spezifikation wird auf die Laufzeitumgebung hin optimiert. Die notwendigen Transformationen lassen sich zum grössten Teil automatisieren. Die Korrektheit der Transformationen garantiert die Konsistenz zwischen Spezifikation und Implementation. Vertreter des operationellen Ansatzes sind z.B. JSD und Paisley [9].

Vergleicht man die beiden Ansätze miteinander, findet man einige charakteristische Unterschiede. Beim *konventionellen* Ansatz ist die Anforderungsspezifikation nichtformal beschrieben. Sie ist für einen technisch nicht vorgebildeten Anwender einfacher verständlich, aber nur sehr schwierig auf ihre Korrektheit, Vollständigkeit und Widerspruchsfreiheit überprüfbar. Die Spezifikation beim *operationellen* Ansatz lässt sich hingegen automatisch prüfen, weil sie formal und deshalb maschinell verarbeitbar ist. Für den technisch nicht vorgebildeten Anwender ist eine solche Spezifikation schwieriger zu verstehen und muss gegebenenfalls in seine Sprache übersetzt werden.

Der konventionelle Ansatz vermischt, im Gegensatz zum operationellen Ansatz, funktionelle und implementationsabhängige Mechanismen. Diese Vermischung entsteht, weil die Aufgliederung des Systems so gewählt werden muss, dass auf der untersten Stufe Code-Module entstehen, die in der Laufzeitumgebung effizient ablaufen. Weiter sind beim konventionellen Ansatz die modulinternen Mechanismen mit der Implementationssprache vermischt, denn die einzige Art, einen Mechanismus zu definieren, besteht darin, ihn zu programmieren. Der operationelle Ansatz spezifiziert die funktionellen Mechanismen unabhängig von der Implementationssprache.

Es scheint, als ob die Unterschiede zwischen den beiden Ansätzen verschwinden würden, wenn man beim konventionellen Ansatz eine 4. Generations-Sprache verwendet. Dies stimmt aber nicht, weil eine operationelle Spezifikation im Gegensatz zum konventionellen Ansatz nicht Lösungen, sondern Probleme beschreibt. Zudem nimmt die operationelle Spezifikation die Laufzeitumgebung nicht vorweg.

Die eben beschriebenen Unterschiede sind natürlich Idealisierungen. Die meisten Software-Entwicklungsmethoden lassen sich nicht scharf dem einen oder anderen Ansatz zuordnen. Vielmehr liegen sie irgendwo dazwischen, mehr zum einen oder anderen Ansatz tendierend, und versuchen die Vorteile beider Ansätze miteinander zu verbinden.

Der operationelle Ansatz hat gegenüber dem konventionellen einige bestechende Vorzüge, er ist aber in der Praxis schwieriger zu verwirklichen. Den meisten Entwicklern macht es Mühe, die formale Spezifikation aus den Anforderungen herzuleiten. JSD ist eine der wenigen Methoden, die auf dem operationellen Ansatz beruhen, den Entwickler schrittweise zur formalen Spezifikation führen und zudem in der Praxis erprobt sind.

Die JSD-Methode

JSD hat drei wesentliche Entwicklungsphasen:

1. Modellierung
2. Netzwerk
3. Implementierung

Die *Modellierungsphase* beantwortet die Frage «Wovon handelt das System?». Das Ziel ist, ein Modell der für das System relevanten Realität zu bilden. Das Modell besteht aus einer Anzahl von praktisch unabhängigen Modellprozessen, welche eine Abbildung der in der Realität vorkommenden Objekte sind.

Bei der *Netzwerkphase* geht es um die Frage «Was soll das System tun?». In dieser Phase werden zu den Modellprozessen schrittweise weitere Prozesse hinzugefügt, welche die Funktionen des Systems festlegen. Dadurch entsteht ein Netzwerk von Prozessen, die über den Austausch von Meldungen oder über Nur-Lese-Zugriffe auf die Daten eines anderen Prozesses miteinander kommunizieren. Das Netzwerk bildet zusammen mit den einzelnen Prozessspezifikationen und der Modellbeschreibung die *Spezifikation* im Sinne des operationellen Ansatzes. Das bedeutet, dass die Spezifikation implementationsunabhängig und (im Prinzip) ausführbar ist.

In der *Implementierungsphase* geht es schliesslich darum, wie das Netzwerk ausgeführt werden soll. Dazu wird die Spezifikation mit Hilfe von Transformationen so umstrukturiert, dass das realisierte System auf die gegebene Hard- und Softwarezielumgebung abgestimmt ist und so die gegebenen Zeitanforderungen erfüllt. Die drei Hauptphasen von JSD sind weiter in einzelne Schritte unterteilt.

Modellierungsphase

In der Modellierungsphase wird als erstes eine Abstraktion der realen Welt gebildet, die für das System von Bedeutung ist, wobei das Schwergewicht auf der zeitlichen Dimension liegt. Diese Abstraktion wird in JSD *Modell* genannt. Das Modell ist im wesentlichen ein Ereignismodell und beschreibt die möglichen Reihenfolgen von Ereignissen. Das Modell hält exakt alle jene Tatsachen fest, die das System kennen muss, um die funktionellen Anforderungen erfüllen zu können. Aus dem Modell werden dann Prozesse abgeleitet, welche das Abbild der Realität innerhalb des Systems darstellen.

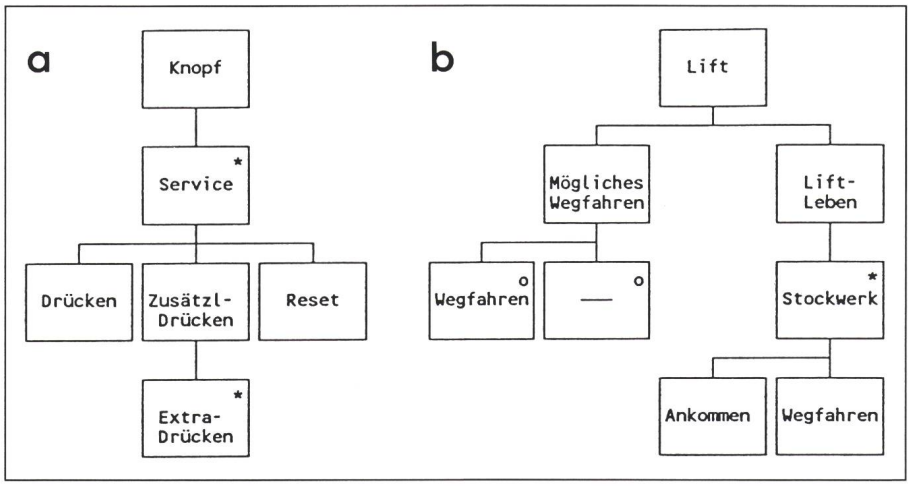
Beschreibung der Realität

Das Modell wird in Form von Ereignissen, Objekten und Rollen (Tab. I) beschrieben. Ein *Ereignis* ist ein atomares Vorkommnis in der realen Welt, über welches das System Informationen benötigt oder Informa-

Ereignisliste für einfaches Liftsystem der Figur 1

Drücken	Ein Knopf wird gedrückt, damit der Lift das zugehörige Stockwerk bedient.
Extra-Drücken	Ein Knopf wird gedrückt, nachdem der Lift schon verlangt wurde, er aber das zugehörige Stockwerk noch nicht bedient hat.
Reset	Der Lift hat das zugehörige Stockwerk bedient.
Ankommen	Der Lift kommt auf einem Stockwerk an, d.h. der entsprechende Stockwerkkontakt spricht an.
Wegfahren	Der Lift verlässt das Stockwerk, d.h. der betreffende Stockwerkkontakt fällt wieder ab.

Tabelle I



Figur 1 Die beiden Rollenstrukturen für das einfachstmögliche Liftsystem
a Rollenstruktur für Liftknopf
b Rollenstruktur für Liftkörper
* Iteration

tionen weitergeben muss. *Objekte* im Sinne von JSD sind Personen, Organisationen, Objekte oder Begriffe, welche eine für das System relevante Abfolge von Ereignissen auslösen oder erleiden. Jedes Objekt hat eine oder mehrere Rollen. Eine *Rolle* beschreibt jeweils ein unabhängiges Verhalten des betreffenden Objekts. Die möglichen Ereignisfolgen einer Rolle werden mit Hilfe eines Strukturdiagramms beschrieben (Fig. 1). Diese Rollenstrukturen definieren alle in der Realität möglichen und deshalb erlaubten zeitlichen Folgen von Ereignissen. Sie sind aus sogenannten Grundstrukturkomponenten aufgebaut (Fig. 2). Alle Ereignisse werden in einer Ereignisliste (Glossar) (Tab. II) beschrieben. Dieses enthält Beschreibungen der Ereignisse selbst sowie der Attribute, die mit ihnen verknüpft sind. *Ereignisattribute* sind Antworten auf Fragen, die über ein Ereignis gestellt werden können, beschreiben also die Merkmale eines Ereignisses. Das Ereignis *Zifferwählen* beim Telefonieren hat z.B. die gewählte Ziffer als Attribut.

Das Modell besteht aus einer Menge nicht miteinander verbundener Modellprozesse. Der einzige Zweck dieser Prozesse ist, die reale Welt im System abzubilden, nichts anderes. Das Modell bildet die Grundlage für die Systemfunktionen, indem es die Begriffe festlegt, mit denen die funktionellen Anforderungen definiert werden müssen. Das Modell definiert somit den Bereich von funktionellen Anforderungen, die es unterstützen kann, und, was noch wichtiger ist, es schliesst diejenigen Anforderungen aus, die es nicht unterstützen kann. Die aktuellen Anforderungen müssen aus diesem Bereich ausgewählt werden, das heisst, das Modell muss umfassend genug sein, um die aktuellen Anforderungen unterstützen zu können.

Modellprozesse

In diesem Schritt werden die Rollenstrukturen zu ausführbaren Modellprozessen erweitert. Dazu müssen zuerst die *Rollenattribute* identifiziert werden. Rollenattribute sind Daten, die wir über ein bestimmtes Objekt abspeichern wollen und bilden die Grundlage für die Datenbank des Systems. Rollenattribute sind private Daten des betreffenden Modellprozesses. Sind die Rollenattribute gefunden, müssen die Mechanismen zur Aktualisierung der einzelnen Attribute spezifiziert werden, indem man geeignete Operationen definiert und diese den zutreffenden Komponenten der Rollenstruktur zuordnet. Die Datentypen der Attribute und die Operationen können dabei entweder nichtformal oder direkt in der gewünschten Programmiersprache beschrieben werden.

Ein Modellprozess beschreibt eine *Prozessklasse,* die mehrere *Exemplare* (instances, Realisierungen) haben kann. Wenn ein Modellprozess mehrere Exemplare hat, muss er einen eindeutigen Identifikator besitzen. Der Identifikator wird in der gleichen Art wie ein Rollenattribut spezifiziert.

Netzwerkphase

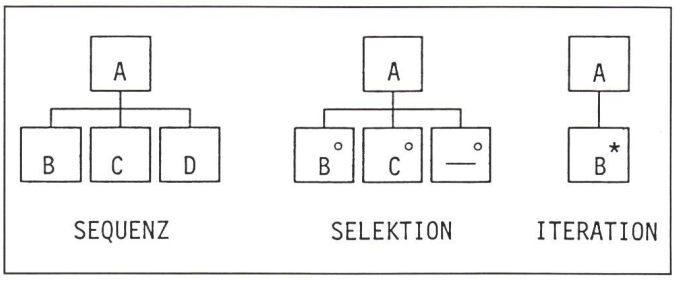
Ausgehend von den untereinander nicht verbundenen Modellprozessen, die aus der Modellierungsphase resultieren, wird schrittweise ein Netzwerk von asynchronen, parallel ablaufenden Prozessen aufgebaut. Es können drei Arten von Prozessen ins Netzwerk eingefügt werden: Informationsfunktionen, interaktive Funktionen und Eingabeprozesse (Fig. 3).

Informationsfunktionen extrahieren Informationen aus dem Modell, um daraus die verlangten Systemausgaben zu erzeugen. Für jede unabhängige Systemausgabe oder für jeden Auslöser einer Systemausgabe wird eine eigene Funktion definiert. Eine Informationsfunktion besitzt als Eingaben sowohl einen Auslöser wie auch Daten, die aus dem Modell stammen. Angestossen wird die Funktion entweder durch ein Ereignis des Modells, durch einen externen Auslöser (Benutzer- oder Hardwareanforderung) oder durch einen periodischen Auslöser (z.B. täglich).

Interaktive Funktionen erzeugen interne Ereignisse. Interne Ereignisse kann man sich wie externe, reale Ereignisse vorstellen, die aber vom System selbst erzeugt werden. Im Gegensatz zu den Informationsfunktionen erzeugen interaktive Funktionen Eingaben ins Modell anstatt Systemausgaben. Oft sind interne Ereignisse notwendig, um ein Modell überhaupt realisieren zu können.

Sind alle Ereignisse eines Systems extern, dann ist das Modell ein reines Abbild der Realität. Werden alle Ereignisse intern erzeugt, ist das System eine Simulation des definierten Modells. Beim Spezifizieren einer interaktiven Funktion muss sich der Ent-

Figur 2 Grundstrukturkomponenten
Sequenz: *A* besteht aus *B*, gefolgt von *C*, gefolgt von *D*
Selektion: *A* besteht entweder aus *B* oder aus *C* oder aus nichts
Iteration: *A* besteht aus 0...*n* Komponenten *B*



wickler überlegen, aus welchen Gründen ein internes Ereignis erzeugt werden soll. Im Gegensatz dazu spielt es keine Rolle, warum ein Ereignis in der realen Welt geschieht. Obwohl ein solches Ereignis die Folge einer Systemausgabe sein kann, wird diese Tatsache beim Definieren des Modells ausser acht gelassen.

Eingabeprozesse sammeln Daten aus der realen Welt (Ereignisse und deren Attribute) und prüfen diese auf Fehler. Sind die Daten korrekt, gibt sie der betreffende Eingabeprozess ans Modell weiter, andernfalls weist er sie zurück und erzeugt eine Fehlermeldung. Die Eingabeprozesse behandeln zwei Arten von Fehlern: *Eingabefehler* treten auf, wenn Eingabedaten nicht ihrer Datentypdefinition entsprechen (Plausibilitätstest). *Kontextfehler* können anhand des Modellzustandes erkannt werden (z.B. unzeitgemässe Ereignisse). Falls ein Ereignis mehreren Rollen zugehört, verteilen die Eingabeprozesse entsprechende Meldungen an die betroffenen Modellprozesse. Eingabeprozesse können auch Dialoge mit dem Benützer enthalten, wenn fehlende Daten zu beschaffen, fehlerhafte zu korrigieren oder mehrere Ereignisse auf einmal zu behandeln sind. Ereignisse aus der realen Welt müssen von den Eingabeprozessen detektiert werden können, sonst lässt sich das Modell im System nicht realisieren.

Für jeden während der Netzwerkphase ins Netzwerk eingefügten Prozess müssen folgende Punkte spezifiziert werden:

- Aufgrund der Informationsbedürfnisse des Prozesses werden die Verbindungen zu den Modellprozessen und möglicherweise zu anderen Prozessen festgelegt. Unter Umständen müssen dazu die betroffenen Modellprozesse erweitert werden. Es gibt zwei verschiedene Arten von Verbindungen (Fig. 4). Die *Datenstromverbindung* ist ein First-in-First-out-Puffer mit konzeptionell unbeschränkter Kapazität. Bei einer *Statusvektorverbindung* inspiziert ein Prozess die privaten Daten, den *Statusvektor*, eines fremden Prozesses, ohne dass dieser davon etwas merkt.

- Mit Hilfe von JSP werden die internen Mechanismen des Prozesses definiert. Die Prozessstruktur wird aus den Daten-Zeit-Strukturen seiner Ein- und Ausgabedatenströme abgeleitet. Mögliche Strukturkonflikte werden durch Aufteilen in Teilprozesse gelöst. Alle

JSD-Begriffe

Modell: Abstraktion der realen Welt, die für das System von Bedeutung ist (Ereignismodell).

Ereignis: Ein atomares Vorkommnis in der realen Welt, worüber das System Informationen braucht oder erzeugen muss. Ein Ereignis kann zu mehreren Rollen gehören.

Ereignisattribut: Informationen, die mit dem Ereignis anfallen und für das System von Bedeutung sind.

Objekt: Person, Organisation, Objekt oder Begriff in der realen Welt, welche eine relevante Folge von Ereignissen auslösen oder erleiden.

Rolle: Ein unabhängiges Verhalten eines Objekts. Ein Objekt kann mehrere Rollen haben. Eine Rolle kann zu mehreren Objekten gehören.

Rollenstruktur: Strukturdiagramm, welches die möglichen Abfolgen der Ereignisse einer Rolle beschreibt.

Rollenattribut: Informationen aus der realen Welt, die vom Modellprozess abgespeichert werden müssen. Sie werden von den Funktionen benötigt, um die verlangten Systemausgaben erzeugen zu können.

Prozess: Ausführung eines sequentiellen Programms auf einem eigenen Prozessor.

Funktion: Eine Menge von Funktionsprozessen, die zusammen Systemausgaben oder interne Ereignisse erzeugen.

Spezifikationsprozess: Modell-, Funktions- oder Eingabeprozess. Jeder Spezifikationsprozess ist ein sequentieller Prozess, der konzeptionell auf einem eigenen Prozessor läuft.

Modellprozess: Prozess im System, der aus einer Rolle entsteht. Seine Struktur beschreibt die möglichen Reihenfolgen von Ereignissen.

Funktionsprozess: Ein Informations-Funktionsprozess erzeugt Systemausgaben, ein interaktiver Funktionsprozess erzeugt systeminterne Ereignisse.

Eingabeprozess: Sammelt und prüft Informationen aus der realen Welt und gibt die korrekten Ereignisse an die Modellprozesse weiter.

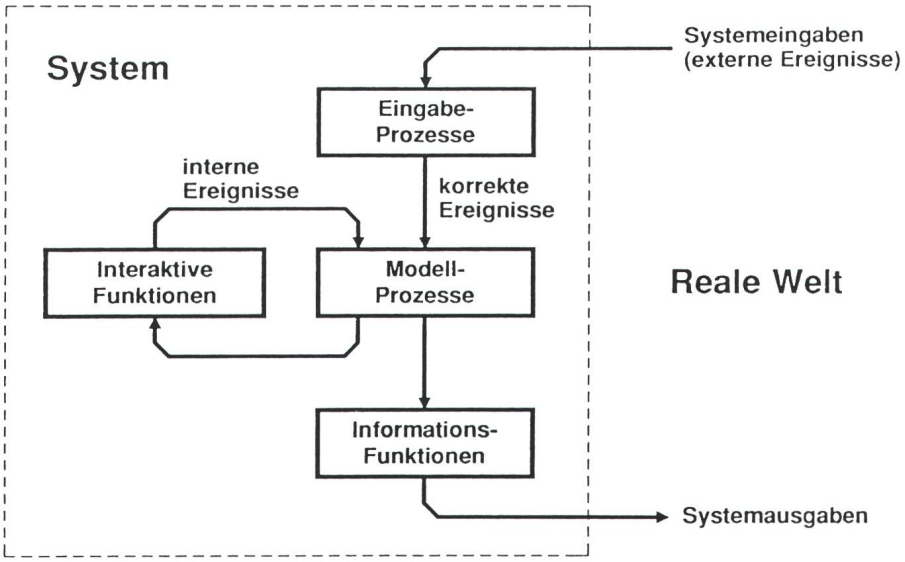
Statusvektor: Private Daten eines Prozesses, die ausschliesslich von ihm aktualisiert werden.

Statusvektorverbindung: Nur-Lese-Zugriff eines Prozesses auf den Statusvektor eines anderen Prozesses.

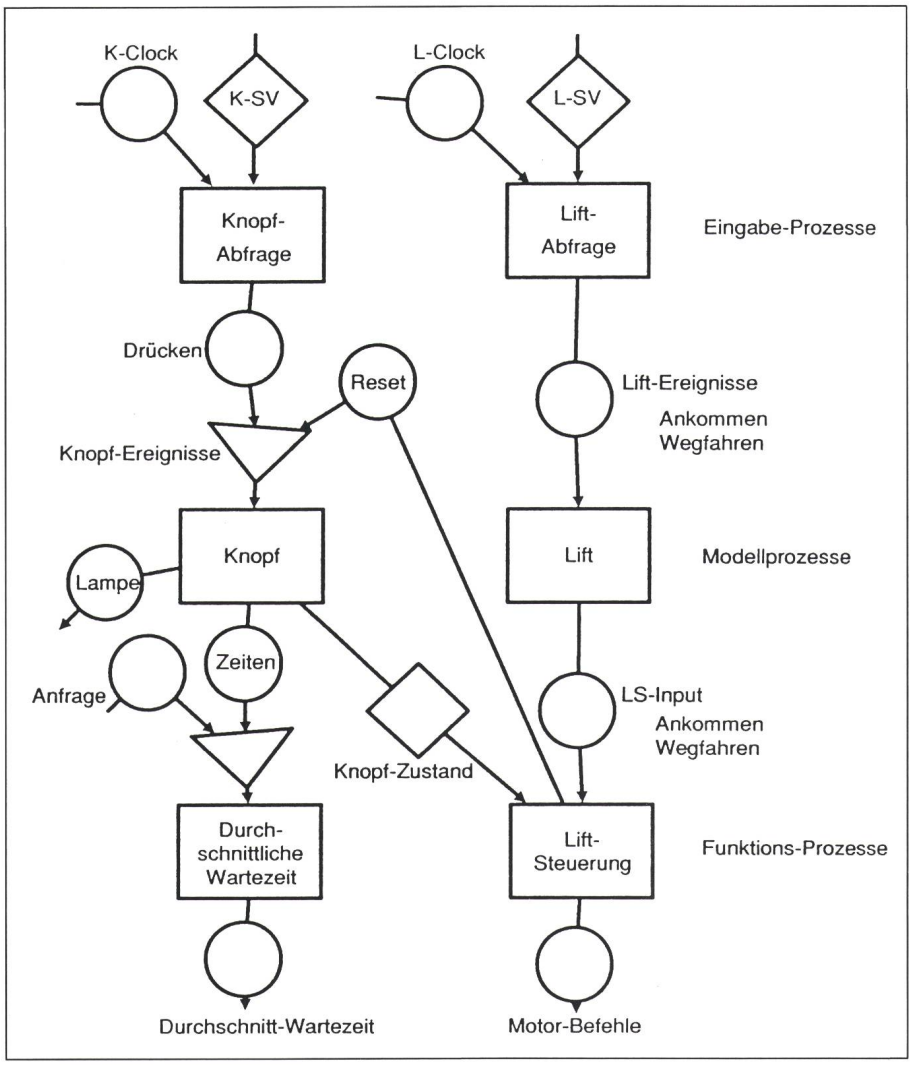
Datenstromverbindung: First-in-First-out-Meldungspuffer mit konzeptionell unendlicher Kapazität.

Implementationsprozess: Eine Menge von Spezifikationsprozessen (oder zergliederten Teilen davon), die unter der Kontrolle eines Ablaufsteuerungsprozesses auf demselben (evtl. virtuellen) Prozessor ablaufen.

Tabelle II



Figur 3 Grundstruktur eines JSD-Systems



Figur 4 Netzwerk-Diagramm eines primitiven Liftsystems

- Prozess(klasse)
- Datenstrom(klasse)
- Verflechtung von Datenströmen
- Statusvektor-Verbindung(sklasse)
- 1:1 Relation
- 1:n Relation
- m:n Relation

notwendigen Operationen werden in einer Operationsliste definiert und den passenden Komponenten der Prozessstruktur zugeordnet. Schliesslich werden die Ablaufbedingungen für die Iterationen und Selektionen in der Prozessstruktur formuliert.

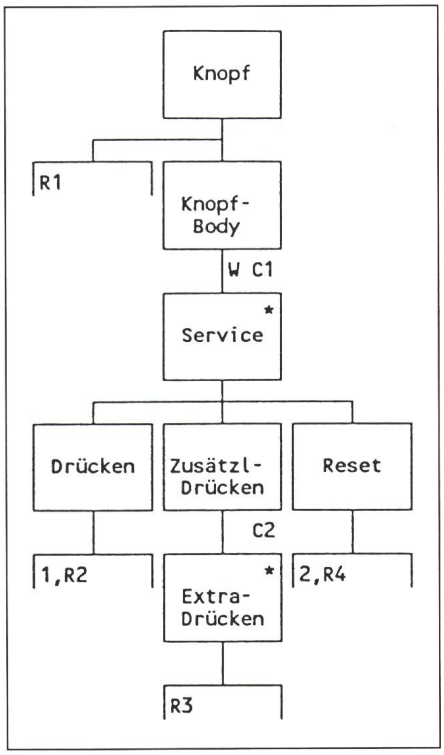
- Die Zeitanforderungen an die Implementation des Prozesses werden definiert, indem man die *Antwortzeit* und die *Aktualität* der produzierten Ausgaben festlegt.

Das Resultat der Netzwerkphase, die *Spezifikation*, ist ein Netzwerk von asynchron miteinander kommunizierenden Prozessen (Fig. 4), deren innere Struktur mit Hilfe von Baumdiagrammen (Fig. 5) festgelegt ist. Jeder Prozess hat seine eigenen privaten Daten, seinen Statusvektor.

Die Spezifikation ist im Prinzip ausführbar. Dazu braucht es entweder einen allgemeinen Scheduler (Ablaufsteuerungsprozess), ein Multitasking-Betriebssystem oder eine Programmiersprache für parallele Prozesse. Weiter müssen die Operationen aller Prozesse in einer geeigneten Programmiersprache definiert sein. Natürlich ist eine solche Ausführung meistens zu wenig effizient, weil die Spezifikation noch viel zu viel Parallelität enthält. Deshalb wird bei der Implementierung die Spezifikation durch Umstrukturierung so optimiert, dass das realisierte System die verlangten Zeitanforderungen erfüllt.

Implementierungsphase

Die Implementierung besteht aus einer Transformation der Spezifikation und der eigentlichen Code-Produktion. Bei der Transformation geht es im wesentlichen darum, zwei Abbildungen zu definieren. Die erste legt fest, wie die Spezifikationsprozesse auf die gegebenen (möglicherweise virtuellen) Prozessoren abgebildet werden (physischer Programmentwurf). Dabei werden einerseits die Ablaufsteuerung des Systems und anderseits die Verbindungen zwischen den Prozessoren definiert. Der Entwickler kann frei entscheiden, welche Teile der Spezifikation mit welcher Priorität ablaufen sollen. Die Abbildung der Spezifikationsprozesse wird in Form eines System-Implementations-Diagramms (SID) dokumentiert (Fig. 6). Das SID legt für jeden Prozessor eine Aufrufhierarchie seiner Prozesse fest.



Figur 5 Modellprozess Knopf
Operations- und Bedingungsliste siehe Tab. III

Operationsliste:

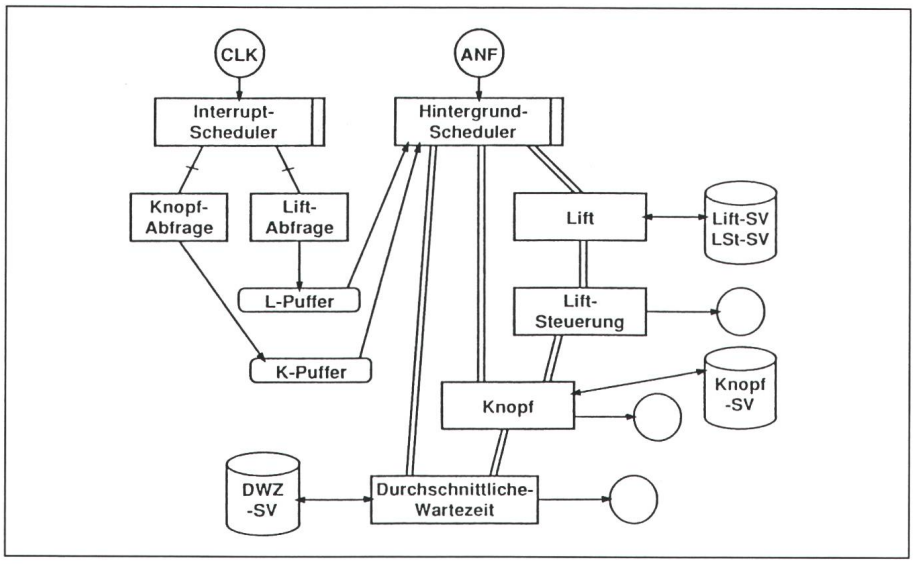
1. [1] Lampe einschalten
2. [1] Lampe ausschalten

Rn. [4] Read *Knopf-Ereignisse*

Bedingungsliste:

C1. [1] true
C2. [1] Ereignis = *Extra-Drücken*

Tabelle III Listen zu Figur 5



Figur 6 System-Implementations-Diagramm eines primitiven Liftsystems

Implementation auf zwei virtuellen Prozessoren

ANF Anfang

CLK Clock

Ablaufsteuerungsprozess (Scheduler, 1 pro Prozessor)

Prozess(teil)

Datenpuffer (FIFO)

Speicher für Statusvektoren (RAM, Disk usw.)

Programm-Inversion

Datenstrom

Vollständige Ausführung eines Programms (Subroutine)

Die zweite Abbildung legt fest, wie die Statusvektoren der Spezifikationsprozesse auf die gegebenen Speicher abgebildet werden (Datenbankentwurf). Gleichzeitig wird festgelegt, wie auf die Statusvektoren zugegriffen wird (Zugriffsmechanismen).

JSD enthält einen Satz von wohldefinierten *Standardtransformationen,* mit welchen die beiden Abbildungen realisiert werden können. Wenn diese Transformationen korrekt angewandt werden, was z.B. durch Automatisierung sichergestellt werden kann, bewahren sie die Korrektheit der Spezifikation. Nachfolgend sind die wichtigsten Transformationen kurz beschrieben.

Die *Programm-Inversion* kombiniert zwei getrennte Spezifikationsprozesse in einen einzigen Implementationsprozess. Der invertierte Prozess wird dabei zu einer Sub-Coroutine des anderen Prozesses. Der invertierte Prozess wird jedesmal dann aufgerufen, wenn der aufrufende Prozess ihm einen Datensatz schicken oder von ihm empfangen will. Der invertierte Prozess fährt dabei an jener Stelle fort, wo er nach dem letzten Aufruf stehengeblieben ist. Sobald er den betreffenden Datensatz konsumiert oder produziert hat, gibt er die Kontrolle wieder an den aufrufenden Prozess zurück.

Bei der *Statusvektor-Abtrennung* wird der Statusvektor jedes Prozessexemplars von seinem Prozesstext getrennt. Dies erlaubt einem Prozessor, mehrere Exemplare einer Prozessklasse zu verwalten. Die verschiedenen Exemplare einer Prozessklasse teilen sich in den gleichen Prozesstext. Beim Aufruf eines Prozesses kann entweder der Prozess selbst oder der aufrufende Steuerprozess den Statusvektor des aufgerufenen Prozesses verwalten.

Die *Programm-Zergliederung* trennt einen Spezifikationsprozess in mehrere Teilprozesse auf, die in verschiedenen Zusammenhängen aufgerufen werden. Damit ist es sogar möglich, verschiedene Teile eines Spezifikationsprozesses in der Implementation mit verschiedenen Prioritäten ablaufen zu lassen.

Bei der *Statusvektor-Zergliederung* wird ein Statusvektor in eine Anzahl Teile aufgetrennt, wobei jeder Teil an einem anderen Ort abgespeichert wird.

JSD im Vergleich mit anderen Ansätzen

Es ist oft schwierig, den Unterschied zwischen verschiedenen Methoden zu verstehen, weil sie meist sehr ähnliche Konzepte verwenden. Ein genaues Verständnis ist aber deshalb wichtig, weil die verwendeten Konzepte einen signifikanten Einfluss auf die Art haben, wie der Entwickler denkt und das zu entwickelnde System versteht. Was noch wichtiger ist, sie beeinflussen die Reihenfolge der Entscheidungen, die der Entwickler trifft. Im folgenden sind die wesentlichsten Unterschiede von JSD zu anderen Methoden und Konzepten kurz zusammengefasst.

Datenflussansatz

Vertreter des Datenflussansatzes sind z.B. Mascot und SASD [7; 8]. Diese Methoden definieren zuerst den Datenfluss des zu entwickelnden Systems. Kontroll- und Ereignisaspekt werden vollständig getrennt betrachtet und erst in einem zweiten Schritt in den Datenfluss eingepasst.

JSD identifiziert in erster Linie Ereignisse und deren mögliche Reihenfolgen. Aus der zeitlichen Ordnung der Ereignisse werden die Kontrollstrukturen abgeleitet. Der Datenfluss des Systems ergibt sich während der Netzwerkphase anhand der Informationsbedürfnisse der einzelnen Funktionen von selbst. Der Datenfluss innerhalb der Prozesse wird durch die zugeordneten Operationen bestimmt. Dadurch erübrigt sich das Abstimmen von Daten- und Kontrollfluss.

Der Datenflussansatz macht keinen Unterschied zwischen Modell und Funktion. SASD stellt zwar die Forderung, dass die Systemstruktur der Problemstruktur entsprechen sollte, betrachtet aber das Abbild der Realität zusammen mit den Systemfunktionen. Im fertigen System ist daher das Modell nicht explizit erkennbar.

JSD verlangt ein explizites Modell als Grundlage für die Systemfunktionen und die Fehlerprüfungen. Damit sind die Begriffe, die später zur Definition der Funktionen gebraucht werden, präzise festgelegt. Weiter lässt sich beim Systemtest auf einfachste Weise prüfen, ob das System mit der Realität Schritt hält. Dies ist um so wichtiger, je komplexer das Modell ist. Da der Entwickler sich zu Beginn des Projekts nur auf das Modell konzentrieren muss, hat er sich mit weniger Informationen gleichzeitig zu befassen

als in SASD. Dies erleichtert den Einstieg wesentlich.

Datenflussdiagramme können hierarchisch, top-down, in Teildiagramme aufgelöst werden. Dadurch entsteht eine funktionale Hierarchie beliebiger Tiefe. Im Gegensatz dazu umfasst eine JSD-Spezifikation nur zwei Ebenen, die Netzwerkebene, welche die Prozesse mit ihren Verbindungen zeigt, und die Prozessebene, welche die innere Struktur der einzelnen Prozesse zeigt. Prozessstrukturen sind zwar hierarchisch, sie entstehen aber aus den Datenstrukturen, nicht etwa aus einer funktionalen Zerlegung. Eine Hierarchie ergibt sich erst bei der Implementation, nämlich die Aufrufhierarchie der Prozesse (oder Teilprozesse) auf einem Prozessor.

Ein JSD-System entsteht nicht top-down, sondern von der Mitte nach aussen, indem schrittweise Prozesse zu den Modellprozessen hinzugefügt werden. Das Problem des Top-Down-Entwurfs liegt darin, dass sich nur dann eine sinnvolle Aufgliederung finden lässt, wenn man das zu entwickelnde System schon kennt. Diese Bedingung ist aber zu Beginn des Projektes gerade nicht erfüllt!

Das Top-Down-Vorgehen zielt direkt auf eine Implementation. Deshalb muss man grössere Teile der Entwicklung nochmals nachvollziehen, wenn man eine andere Implementation wünscht. Bei JSD kann man einfach auf die Spezifikation zurückgreifen und neue Transformationen wählen. Dies ist mit kleinem Aufwand möglich, wenn man geeignete Werkzeuge verwendet. Zudem ist sichergestellt, dass sich dabei die Funktionalität des Systems nicht verändert.

Datenmodellierung

Die verschiedenen Arten der Datenmodellierung (Datenbankentwurf) befassen sich alle mit einer Kombination von Entitäten, Attributen und Relationen [10]. Das Ziel ist ähnlich wie bei der JSD-Modellierung. Das Datenmodell soll die Grundlage sein für den Rest des Systems. Diese Grundlage ist stabiler als die funktionellen Anforderungen und dient als Kommunikationsmittel für den Dialog mit dem Anwender.

JSD liefert direkt die Grundlage für die Datenmodellierung. Entitäten und Attribute sind in einem JSD-Modell explizit enthalten, Relationen nur implizit. Fremdschlüssel in einem Modellprozess, d.h. Attribute von Modellprozessen, welche Identifikatoren von anderen Modellprozessen sind, können als (abgemagerte) Relationen interpretiert werden. Geht man davon aus, dass jeder Prozess-Statusvektor ein eigener Datenbanksatz ist, entsteht ein erstes, einfaches Datenmodell. Dieses lässt sich verfeinern und der gewählten Implementation besser anpassen, indem man z.B. mehrere Statusvektoren in einen Datenbanksatz zusammenlegt oder einen Statusvektor auf mehrere Datenbanksätze aufteilt.

Beginnt man die Entwicklung mit dem Datenmodell, muss man in einem weiteren Schritt die Ereignisse definieren, welche die Datenbank aktualisieren und die zugehörigen Konsistenzregeln spezifizieren. In JSD geht man den umgekehrten Weg. Zuerst kommen die Ereignisse und deren mögliche Reihenfolgen. In den Modellprozessen wird exakt die Logik zur Aktualisierung der Rollenattribute definiert. Zusammen mit den Modellprozessstrukturen ist damit automatisch die Konsistenz der Datenbank sichergestellt. Die Konsistenzregeln müssen also nicht nachträglich auf die Datenbank aufgepfropft werden. Im Gegensatz zu einem Datenmodell kann mit einem JSD-Modell das dynamische Verhalten des Systems detailliert diskutiert werden. Für die meisten Anwender ist es einfacher, ein Modell anhand der Ereignisse und ihrer Reihenfolgen zu diskutieren, als anhand eines statischen Datenmodells. Für Systeme ohne grosse Dynamik ist die Datenmodellierung möglicherweise besser geeignet.

Zusammenfassend kann gesagt werden, dass die Datenmodellierung eine wesentliche Ergänzung zu JSD ist, wobei die Grundlage des Datenmodells von JSD geliefert wird.

Objektorientierter Ansatz

Der objektorientierte Ansatz (OOA) ist ein äusserst wirkungsvolles Konzept zur Strukturierung von Software. Er ist aber noch keine volle Methode. JSD gibt im Vergleich dazu wesentlich mehr Führung beim Entwicklungsprozess. In JSD ist nur die Modellierungsphase objektorientiert. Die Funktionen werden in der Netzwerkphase nicht notwendigerweise wegen ihrer Objektorientierung ausgewählt [11].

Der objektorientierte Ansatz betrachtet zuerst die Objekte und erst in zweiter Linie die Operationen (bzw. Methoden im Sinne des OOA). Bei JSD ist es genau umgekehrt. Die Erfahrung zeigt, dass der JSD-Ansatz, vor allem bei schwierigeren Fällen, leichter ist. Ein weiterer Unterschied zum OOA besteht darin, dass JSD bei der Modellierung nur Ereignisse betrachtet, die den Modellzustand verändern. Ein JSD-Objekt ist einschränkender definiert als ein OOA-Objekt. Nur Objekte, die eine zeitliche Ereignisfolge erleiden oder über welche Daten gespeichert werden müssen, sind in JSD Objekte. Deshalb führt JSD auch auf weniger Objekte als der OOA [11].

Wie der Datenflussansatz, macht auch der OOA keine Unterscheidung zwischen dem Modell der Realität und den funktionellen Anforderungen, d.h. ein Objekt im Sinne des OOA beschreibt beides miteinander.

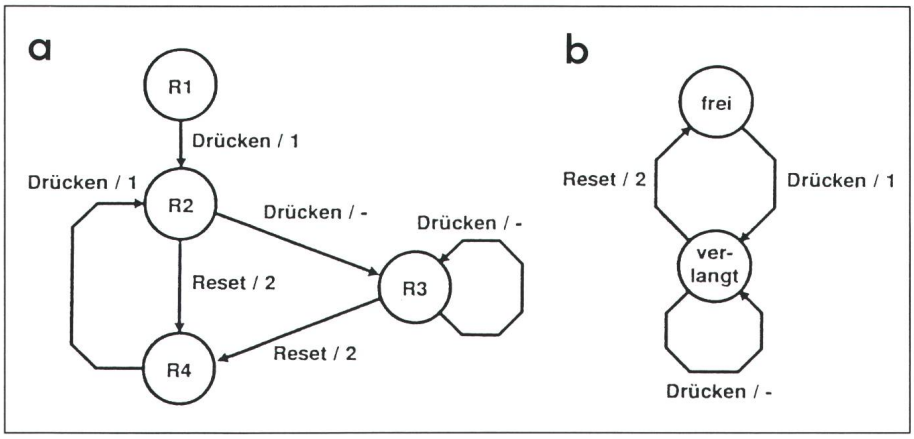
Der OOA ist sehr gut für eine Bottom-Up-Entwicklung geeignet. Von da her ist der OOA eine gute Ergänzung zu JSD, wenn es darum geht, die Operationen innerhalb eines Funktionsprozesses zu definieren, insbesondere bei Standardoperationen, die in verschiedenen Prozessen gebraucht werden.

Im Gegensatz zum OOA ist JSD nicht gut geeignet für abstrakte, mathematische Objekte. JSD-Objekte sind auch nicht wiederverwendbar, weil sie genau auf die Realität zurechtgeschneidert sind. Hingegen lässt sich die Vererbung von Verhalten mit Hilfe von Rollen, die zu mehreren Objekten gehören, realisieren.

Zustandsdiagramme und Baumstrukturen

Zustandsdiagramme werden häufig für die Entwicklung von Hard- und Softwaresystemen eingesetzt, vor allem im technischen Bereich. Sie gestatten es, beliebige Strukturen zu definieren und sind einfach verständlich. Jedes Baumdiagramm kann als Zustandsdiagramm dargestellt werden. Die Figur 5 zeigt den Modellprozess für den Liftknopf, wie er in Figur 1 definiert ist. Die zusätzliche Komponente Knopf-Body braucht es, weil durch Hinzufügen der Leseoperation R1 eine Sequenz entsteht. Figur 7 zeigt das zu Figur 5 äquivalente Zustandsdiagramm. Jede Lese-Operation (*Rn*) entspricht einem Zustand in Figur 7a. Die Zustandsübergänge entsprechen den Ereignissen mit ihren zugeordneten Operationen.

Das Zustandsdiagramm kann vereinfacht werden, indem die Zustände R1 und R4 in den Zustand «frei» und die Zustände R2 und R3 in den Zu-



Figur 7 Zustandsdiagramme

a Äquivalentes Zustandsdiagramm zu Figur 6
R1 Systemstandzustand, in den man nie zurückkehrt
b Vereinfachtes Zustandsdiagramm

stand «verlangt» zusammengelegt werden (Fig. 7b). Das vereinfachte Zustandsdiagramm erlaubt die gleiche Folge von Ereignissen wie das Baumdiagramm, enthält aber weniger Information. Der Begriff *Service,* als logische Gruppierung von Ereignissen, fehlt im Zustandsdiagramm. Das Baumdiagramm legt genau fest, was ein «Service» ist und ermöglicht dadurch eine exaktere Spezifikation von Funktionen, die auf diesem Begriff basieren. Beim Zustandsdiagramm müssen solche Begriffe mit zusätzlichen Mitteln festgelegt werden, was weniger übersichtlich und zuverlässig ist. Zudem wird der Entwickler beim Baumdiagramm dazu angeleitet, sich die verwendeten Begriffe genau zu überlegen.

Beim Entwickeln von Systemen fördern Zustandsdiagramme dynamisches Denken (was muss ich tun, wenn im Zustand *X* das Ereignis *Y* auftritt?). Baumdiagramme führen den Entwickler eher zu einer statischen Sichtweise (wie sind die Ereignisse logisch geordnet?) und ermöglichen dadurch ein tieferes Verständnis. Insbesondere zeigen Baumdiagramme die Geschichte eines Objekts oder Prozesses. Für Prozesse, bei denen die Geschichte eine untergeordnete Rolle spielt, dürften Zustandsdiagramme besser geeignet sein.

Werkzeuge

Eine Methode ist vor allem bei komplexeren Systemen nur praktikabel, wenn sie durch einen Satz von Werkzeugen unterstützt ist. Langsam setzt sich auch die Erkenntnis durch, dass Werkzeuge auf einer Methode aufbauen sollten, nicht umgekehrt! Zu JSD gibt es einen Satz von Werkzeugen, welche die Modellierung, die Netzwerkphase und die Code-Produktion unterstützen. Diese Werkzeuge sind genau auf JSD zugeschnitten und erlauben es, direkt aus der Spezifikation ausführbaren Code und die Dokumentation zu erzeugen (z.B. die Figuren 1, 4 und 5 wurden damit erzeugt). Die Werkzeuge überprüfen online die Konsistenz der Spezifikation.

Zu JSD gibt es auch eine passende Projektmanagementmethode, die aber bis jetzt noch nicht werkzeugunterstützt ist.

Erfahrungen mit JSD

Die Erfahrung zeigt, dass die konsequente Anwendung von JSD ein vertieftes Verständnis sowohl des zu entwickelnden Systems wie auch des Entwicklungsvorganges bewirkt. Die klare und knappe Dokumentation, ein zwingendes Nebenprodukt der Entwicklung, ist ein ausgezeichnetes Mittel zur Kommunikation und Teamarbeit. Schwierigkeiten ergeben sich vor allem aus dem zu anderen Verfahren unterschiedlichen Denkansatz. Die meisten Entwickler sind sich gewohnt, in Implementationen zu denken, was die Implementationsunabhängigkeit der Spezifikation gefährden kann. Diese Schwierigkeit ist bei entsprechender Praxis und guter Betreuung durch erfahrene JSD-Entwickler ohne weiteres zu überwinden.

Als Schlussfolgerung kann gesagt werden, dass JSD ein gangbarer und praktisch erprobter Weg ist, den operationellen Ansatz mit all seinen Vorteilen in die Praxis umzusetzen.

Literatur

[1] *M. A. Jackson:* System development. Englewood Cliffs/N.J., Prentice-Hall, 1983.
[2] *J. R. Cameron:* JSP and JSD: The Jackson approach to software development. IEEE-Tutorial 516. Silver Spring/MD, IEEE Computer Society Press, 1983 (second edition 1989).
[3] *J. R. Cameron:* An overview of JSD. IEEE Trans. Software Engineering SE-12(1986)2, p. 222...240.
[4] *J. R. Cameron:* The modelling phase of JSD. Information and Software Technology 30(1988)6, p. 373...383.
[5] *A. Renold:* Jackson system development for real time systems. Scientia Electrica 34(1988)2, p. 3...44.
[6] *A. Renold:* Designing a music synthesizer with the JSD method. Scientia Electrica 34(1988)4, p 3...46.
[7] *T. DeMarco:* Structured analysis and system specification. Yourdon Press, New York, 1978.
[8] *E. Yourdon* and *L. L. Constantine:* Structured design: fundamentals of a discipline of computer program and systems design. Englewood Cliffs/N. J., Prentice-Hall, 1978.
[9] *P. Zave:* The operational versus the conventional approach to software development. Communications of the ACM (Association for Computing Machinery) 27(1984)2, p. 104...118.
[10] *M. Vetter:* Strategie der Anwendungssoftware-Entwicklung, Planung, Prinzipien, Konzepte. Stuttgart, Teubner, 1988.
[11] *A. Birchenough* and *J. R. Cameron:* JSD and Object-oriented design. SI-Informationen 15(1989)23, p. 7...11