Universität Hamburg
Fachbereich Informatik

Diplomarbeit

Modellierung und Validierung dynamischen Systemverhaltens am Beispiel von Tycoon und StP/OMT

Betreuer:
Prof. Dr. Joachim W. Schmidt
Technische Universität Hamburg-Harburg
Forschungsschwerpunkt 4
Arbeitsbereich Softwaresysteme

Prof. Dr. Rüdiger Valk
Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Theoretische Grundlagen der Informatik

vorgelegt von:
Oliver Nietsch
Große Straße 56 b
21075 Hamburg
Mat.-Nr.: 401 04 66

Hamburg, den 18. April 1997

Meinen Eltern

Bei der Erstellung dieser Arbeit standen mir viele Menschen zur Seite, bei denen ich mich herzlich bedanken möchte.

Mein spezieller Dank gilt Herrn Prof. Dr. Joachim W. Schmidt und Herrn Prof. Dr. Rüdiger Valk sowohl für die Unterstützung und das besondere Interesse am Thema dieser Arbeit als auch für die Anregungen während der Arbeitsdurchführung.

Herrn Gerald Schröder danke ich für seine unbegrenzte Geduld bei der Beantwortung aller meiner Fragen während des gesamten Zeitraums der Arbeitserstellung. Unsere zahlreichen Diskussionen zum Thema haben mir auf meinem Weg zum Abschluß dieser Arbeit stets weitergeholfen.

Nicht zuletzt gilt mein Dank Herrn Daniel Moldt, der mich vor allem bezüglich meiner Fragestellungen im Bereich der theoretischen Grundlagen dieser Arbeit immer auf neue zu berücksichtigende Aspekte hinwies.

Meinen Eltern möchte ich an dieser Stelle dafür danken, daß sie mir meinen Lebensweg ermöglicht haben und mir auch im Rahmen dieser Arbeit nicht nur als unermüdliche Korrekturleser jederzeit zur Seite standen.

Hamburg, den 18. April 1997 Oliver Nietsch

Typographie

In dieser Arbeit werden unterschiedliche Schriftarten verwendet, um die Lesbarkeit zu verbessern. Die Bedeutung der einzelnen Schriftarten ist in der folgenden Tabelle wiedergegeben. Darin bezeichnen „sonstige Firmen- oder Produktnamen“ von der Firma Interactive Development Environments (IDE) und ihrer Produktfamilie Software through Pictures (StP) verschiedene Unternehmen oder kommerzielle Produkte.

Schriftart	**Bedeutung**	**Beispiel**
Emphasized	Englische Begriffe, Mathematische Formeln	*Software Engineering* $I \cup generated(E)$
Sans Serif	Sonstige Firmennamen, Produktnamen	Star Division Unix, StarView
`Typewriter`	Systemkommandos, C Ausdrücke	`LD_LIBRARY_PATH=...` `char *sig = "Name"`
Slanted	Tycoon Language Ausdrücke	*omsRepository.check(rep)*
Slanted	Reservierte Tycoon Language Schlüsselworte	***let*** a = *3*

Für die Darstellung der Tycoon Language (TL) Implementierungsdetails gelten die gültigen Tycoon *Layout*- und Namenskonventionen wie in [Mathiske et al. 93] angegeben.

Inhaltsverzeichnis

1. Einleitung **1**

1.1 Motivation 1

1.2 Problemstellung 4

1.3 Zielsetzung 7

1.4 Struktur der Arbeit 8

2. Werkzeugorientierte Systementwicklung **11**

2.1 Computerunterstützte Anwendungsentwicklung 11

2.1.1 Anforderungen an CASE 11

2.1.2 Werkzeugintegration 12

2.2 Objektorientiertes Modellieren und Entwerfen mit OMT 14

2.2.1 OMT Metamodell 14

2.2.2 Modellierung dynamischer Systemaspekte 16

2.3 Die CASE Umgebung Software through Pictures (StP/OMT) 18

2.3.1 Architekturmerkmale 19

2.3.2 Basisdienste des StP Kerns 20

2.3.3 Das StP Repository 21

2.3.4 Anwendungssicht auf das StP Repository 23

2.3.5 Zugriffsmöglichkeiten auf StP Repositoryobjekte 25

3. Anbindung des Tycoon Systems an die StP Umgebung **27**

3.1 Basisfunktionalität der StP Umgebung 28

3.2 Abbildung der StP OMS Funktionalität auf die Tycoon Ebene 29

3.2.1 Tycoon C Calls für den StP API Funktionsaufruf 29

3.2.2 Abbildung von StP API Operationen auf Tycoon Funktionen 30

3.3 Die Tycoon Umgebung StPEnv für den StP Repositoryzugriff 31
3.3.1 Modellierungsaspekte 32
3.3.2 Bibliotheksumfang 33
3.3.3 Grenzen der Funktionalität 34

4. Tycoon Statecharts zur Modellierung dynamischen Systemverhaltens 37
4.1 Statecharts als strukturierte endliche Automaten 38
4.2 Zustände und Zustandsverfeinerungen 42
4.2.1 Basiszustand 45
4.2.2 Hierarchischer Zustand 46
4.3 Zustandsübergänge 51
4.3.1 Ereignis 52
4.3.2 Aktion 53
4.4 Semantik von Ausführungsschritten 54
4.4.1 Systemkonfiguration 55
4.4.2 Schrittkonstruktion 56

5. Validierung dynamischen Systemverhaltens 59
5.1 Einordnung in den Softwareentwicklungsprozeß 60
5.2 Validierungstechniken 63
5.2.1 Generierung validierbarer Prototypen 64
5.2.2 Anbindung externer Dienste 68
5.3 Vorgehensweisen zur modellbasierten Verhaltensvalidierung 70
5.3.1 Statische Validierung 71
5.3.2 Dynamische Validierung 71
5.4 Tycoon Modellierungs- und Validierungsumgebung 73
5.4.1 Umgebungsumfang 74
5.4.2 Umgebungsarchitektur 75

6. Zusammenfassung und Ausblick 77

A. Anbindung der StP API Funktionen an das Tycoon System 81
A.1 Generierungsskript für dynamische StP Bibliothek lipstp.so 81
A.2 Pfade zu den StP Systemdateien 81
A.3 StP Systemumgebungsvariablen 81
A.4 Bindung und Verwendung der StP API Funktionalität 82

B. Tycoon Schnittstellen 83
B.1 Modellierungs- und Validierungsumgebung StatechartEnv 83
B.1.1 Modul Statechart 83
B.1.2 Modul StP2Tycoon 85
B.2 StP Repository Gateway StPEnv 86
B.2.1 Module mit spezieller Tycoon Funktionalität 86
B.2.2 Module mit spezieller StP PDM Datentyp Funktionalität 86
B.2.3 Module mit allgemeiner StP PDM Datentyp Funktionalität 87

C. Anwendungsbeispiel Television 89
C.1 Statecharts 89
C.1.1 Zustand Television 89
C.1.2 Verfeinerung Zustand Image 90
C.2 TL Repräsentation 91
C.3 Szenariobasierter Validierungslauf 96

Literaturverzeichnis 97

Abbildungsverzeichnis

1.1 Produktivitätsdeterminanten eines Softwareentwicklungsprojekts 2
1.2 Unterstützung des Softwareentwicklungsprozesses 4
1.3 Integration, Erweiterung und Nutzung generischer Dienste in Tycoon 6

2.1 Integrierte CASE Umgebung 13
2.2 OMT Modelle 15
2.3 OMT Ereignisflußdiagramm 16
2.4 OMT Ereignispfaddiagramm 17
2.5 StP/OMT Architektur 19
2.6 StP Persistent Data Model (PDM) 22
2.7 Integritätsbedingungen zwischen StP PDM Datentypen 23
2.8 Anwendungssicht auf das StP Repository 23
2.9 Abbildung von StP Anwendungsdatentypen auf PDM Datentypen 24

3.1 Anbindung des Tycoon Systems an die StP Umgebung 27
3.2 StP API Funktionsaufruf vom Tycoon System 32
3.3 Tycoon StP Bibliothek *StPEnv* 33

4.1 Statechart Metamodell 39
4.2 *Television* Statechart 41
4.3 Generalisierungseigenschaften von Zuständen 42
4.4 Verfeinerter Zustand *Power_On* 43
4.5 ODER-Zustand *Videotxt* 46
4.6 UND-Zustand *Display* 49
4.7 Ausgangskonfiguration 55
4.8 Folgekonfiguration 55

5.1 Systemanalyse durch Validierung 59
5.2 Einordnung der Validierung in den Modellierungsprozeß 61
5.3 Validierungszyklus 62
5.4 Modellbasiertes Prototyping 64
5.5 Anbindung externer Dienste 68
5.6 Modellierungs- und Validierungsumgebung 74
5.7 Umgebungsarchitektur 76

1. Einleitung

Die kommerzielle Entwicklung von großen Softwaresystemen ist dadurch gekennzeichnet, daß Anwendungsentwicklungsprojekte mit einer Vielzahl von Mitarbeitern ohne eine der Softwareerstellung zugrundeliegende Entwicklungsmethode und ohne die Unterstützung durch computerbasierte Entwicklungswerkzeuge kaum noch durchzuführen bzw. zu handhaben sind [Sommerville 96]. Die Softwareentwicklung bedingt heutzutage mehr denn je den Einsatz leistungsfähiger und integrierter Entwicklungswerkzeuge, um die Forderungen nach einer hohen Softwarequalität und kurzen Entwicklungszyklen befriedigen zu können.

Im Rahmen dieser Arbeit wird ein methodisches Vorgehen zur Anwendungsentwicklung (*Object Modeling Technique* OMT [Rumbaugh et al. 91]) und dessen Werkzeugunterstützung zur computerbasierten Systemspezifikation betrachtet. Der Schwerpunkt liegt hierbei auf der Spezifikation des dynamischen Systemverhaltens. Untersucht wird, wie dynamisches Systemverhalten am Beispiel eines ereignisgesteuerten Systems modelliert und seine Prozeßspezifikation zur Analyse des Systemverhaltens schon in einer frühen Phase des Softwareentwicklungsprozesses validiert werden kann. Die Werkzeugunterstützung erfolgt mit Hilfe der *Computer Aided Software Engineering* (CASE) Umgebung Software through Pictures (StP) und dabei speziell mit dem CASE Werkzeug StP/OMT der StP Produktfamilie und der integrierten Programmierumgebung Tycoon[1] des Arbeitsbereichs „Softwaresysteme" an der Technischen Universität Hamburg-Harburg.

Durch die Validierung sollen nachträgliche Konzeptänderungen und kostenintensive Fehlerbehebungen in späteren Projektphasen, z.B. der Implementierung, vermieden werden [Leveson et al. 94], um die in Abschnitt 1.1 dargestellten Produktivitätsdeterminanten eines Softwareentwicklungsprojekts zu erfüllen.

1.1 Motivation

Der Erfolg eines Softwareprojekts wird durch den aufeinander abgestimmten Einsatz der verwendeten Entwicklungswerkzeuge und der am Projekt beteiligten Mitarbeiter auf der Basis einer dem Softwareentwicklungsprozeß zugrundeliegenden Entwicklungsmethode bestimmt

[1] Tycoon: Typed Communicating Objects in Open Environments

[Balzert 93]. Aber nicht nur diese den Softwareentwicklungsprozeß bestimmenden Faktoren eines Anwendungsentwicklungsprojekts sind ausschließlich für die erfolgreiche Durchführung eines Softwareprojekts maßgebend. Vielmehr haben neben der operationalen Durchführung eines Softwareentwicklungsprozesses innerhalb eines Anwendungsentwicklungsprojekts als Resultat der Interaktion zwischen Mitarbeitern, verwendeten Werkzeugen und einer zugrundeliegenden Entwicklungsmethode auch die in der Abbildung 1.1 dargestellten Determinanten der Projektproduktivität [Oberweis 96] einen Einfluß auf die Prozeßgestaltung und somit auf den Gesamterfolg eines Softwareentwicklungsprojekts. Produktivitätsdeterminanten wirken als Einschränkungen oder Anforderungen auf den Anwendungserstellungsprozeß ein.

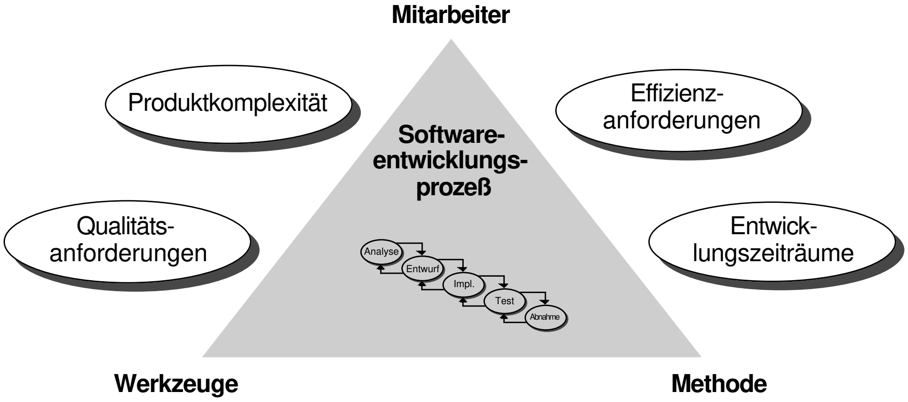


Abbildung 1.1: Produktivitätsdeterminanten eines Softwareentwicklungsprojekts

Eine zunehmende Produktkomplexität bei steigenden Effizienzanforderungen an die Systementwicklung innerhalb verkürzter Entwicklungszeiträume kann nur durch eine optimale Gestaltung des Systementwicklungsprozesses und seiner optimalen Werkzeug- und Methodenunterstützung befriedigt werden. Desweiteren sind steigende Qualitätsanforderungen zu erfüllen, die nicht nur einen Abnahmetest vorsehen, sondern so zu berücksichtigen sind, daß schon in den frühen Phasen des Softwarelebenszyklus Qualität in das zu entwickelnde Anwendungsprodukt eingebracht wird. Eine vom Auftraggeber geforderte Produktzertifizierung nach DIN EN ISO 9001 stellt noch weitere Qualitätsanforderungen an den Softwareerstellungsprozeß [Versteegen 96]. Aus diesen Gründen ist es auf der Seite der Anwender und Entwickler erforderlich, bereits in der Analysephase des Anwendungsentwicklungsprozesses eine klare und vollständige Vorstellung über das zu erstellende System zu erhalten.

Zur Berücksichtigung der in Abbildung 1.1 wiedergegebenen Produktivitätsdeterminanten eines Softwareentwicklungsprojekts werden die folgenden Gestaltungsmöglichkeiten des Softwareentwicklungsprozesses betrachtet:

- Grundlage eines Softwareentwicklungsprozesses ist ein methodenbasiertes Vorgehen über den gesamten Zeitraum des Softwarelebenszyklus von der anfänglichen Problemformulierung bis zur Implementierung[2] [Raasch 91]. Die Spezifikation des zu erstellenden Anwendungssystems erfolgt auf der Basis formaler Modelle, wobei verschiedene Sichten auf das zu erstellende System zu spezifizieren sind. Die in Abschnitt 2.2 vorgestellte objektorientierte Anwendungsentwicklungsmethode OMT umfaßt hierfür ein Objektmodell, ein funktionales Modell und ein dynamisches Modell zur Spezifikation des Verhaltens eines zu entwickelnden Anwendungssystems.

Unterstützt werden soll die Modellierung der dynamischen Systemaspekte mit Hilfe von grafischen Werkzeugen, welche ein verbessertes Systemverständnis auf seiten der Anwender und Entwickler durch grafische Abstraktion, Vereinfachung und eine verbesserte Überschaubarkeit der modellierten Systemaspekte erzielen. Auf diese Weise kann eine steigende Produktkomplexität für alle am Prozeß Beteiligten handhabbar werden. Die eingesetzten grafischen Werkzeuge müssen, wie in Abschnitt 2.1.2 dargestellt, in einer integralen Art miteinander verknüpft werden, um eine effiziente Teamorganisation innerhalb eines Softwareprojekts zu realisieren.

- Resultierend aus den oben dargestellten Qualitätsanforderungen an den Softwareentwicklungsprozeß müssen Verifikationsmöglichkeiten zur Überprüfung der formalisierbaren Eigenschaften der einzelnen Modelle in den Entwicklungswerkzeugen vorhanden sein.

- Eine weitere Softwarequalitätssteigerung soll mit Hilfe der Validierung der nicht formalisierbaren Modelleigenschaften durch den Anwender erreicht werden. Die Zielsetzung ist hierbei, den Anwender in den in Abschnitt 5.1 dargestellten Validierungszyklus innerhalb der Analysephase mit einzubeziehen, um im Rahmen der Validierung der Prozeßspezifikation eines zu erstellenden Systems das spezifizierte Systemverhalten gegen die Erwartung des Anwenders zu untersuchen. Die Funktionen eines Systems werden auf der Basis des dynamischen Systemmodells durch den Anwender getestet und können bei Bedarf umgehend in der Spezifikation geändert werden.

  Dieses Vorgehen zur Verhaltensanalyse wird in die dem Softwareentwicklungsprozeß zugrundeliegende Entwurfsmethode integriert. Somit gilt die Validierung dynamischen Systemverhaltens als Ergänzung zur statischen und dynamischen Modellierung von Anwendungssystemen innerhalb der Spezifikationsphase von Softwareprojekten.

  Desweiteren werden Entwicklungszeiträume verkürzt, wenn im Rahmen von modellbasiertem *Prototyping* (vgl. Abschnitt 5.2.1) ein Validierungsprototyp automatisch aus der Spezifikation seines dynamischen Systemverhaltens generiert wird, der als Validierungsgrundlage für den Anwender dient.

Im Rahmen dieser Arbeit wird die OMT methodenbasierte, werkzeugorientierte Modellierung und Validierung dynamischen Systemverhaltens und das in Abschnitt 5.1 dargestellte

[2] Die Phasen Test und Wartung werden nicht berücksichtigt, da der Schwerpunkt dieser Arbeit in der Betrachtung der frühen Phasen des Softwarelebenszyklus (Analyse und Entwurf) liegt.

Einbringen der Validierungsergebnisse in den Softwareentwicklungsprozeß als eine Form eines Anwendungserstellungsprozesses untersucht.

1.2 Problemstellung

Wie in Abschnitt 1.1 dargestellt, soll die Spezifikation des dynamischen Systemverhaltens innerhalb des Softwareentwicklungsprozesses werkzeugorientiert vorgenommen werden. Desweiteren sollen die verwendeten Entwicklungswerkzeuge Möglichkeiten zur Verifikation und Validierung der dynamischen Systemaspekte bieten.

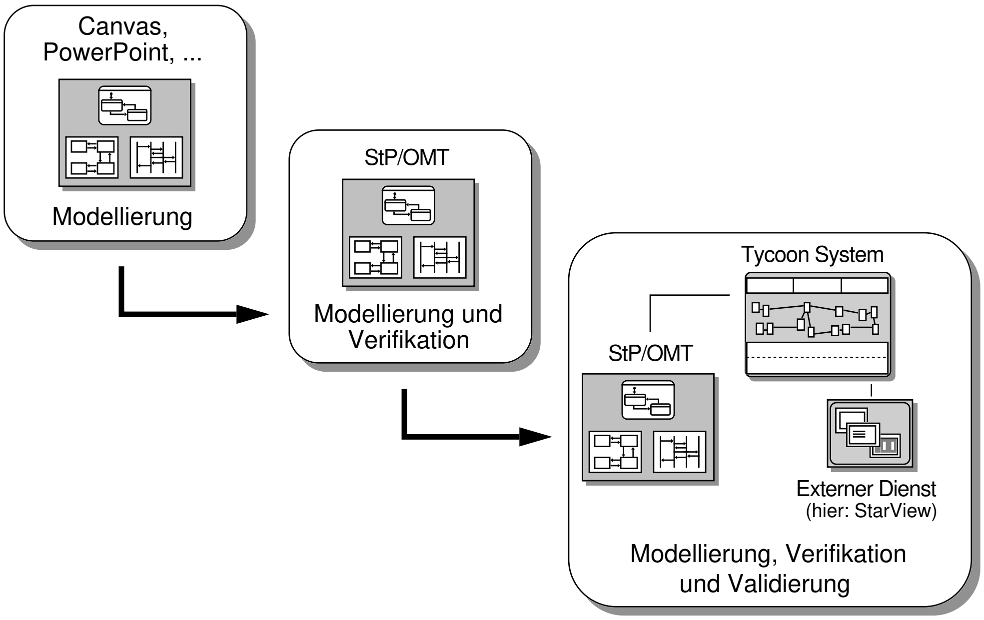


Abbildung 1.2: Unterstützung des Softwareentwicklungsprozesses

In der Abbildung 1.2 sind die einzelnen Formen einer werkzeugunterstützten Spezifikation dynamischen Systemverhaltens dargestellt. Dabei führt der Pfad der Werkzeugunterstützung von isolierten, methodenunabhängigen und computerbasierten Modellierungswerkzeugen (z.B. einfache Zeichenwerkzeuge wie Canvas oder Microsoft PowerPoint) zu einer in Abschnitt 5.4 dargestellten, integrierten, OMT methodenorientierten und computerbasierten Umgebung zur Modellierung, Verifikation und Validierung dynamischen Systemverhaltens. Innerhalb dieser Umgebung wird dem Anwendungsentwickler als grafisches *Front End* zur Modellierung und Verifikation das CASE Werkzeug StP/OMT zur Verfügung gestellt. Die Validierungsoperationen werden innerhalb des Tycoon Systems realisiert, welches, wie in Kapitel 3 dargestellt, an die CASE Umgebung Software through Pictures über das StP Repository angebunden wird.

Eine Verbesserung der aus Sicht eines komplexen Softwareerstellungsprozesses primitiven Entwicklungswerkzeuge zur ausschließlich grafischen Visualisierung von Modellierungsdetails einer Systemspezifikation bietet das, innerhalb dieser Arbeit betrachtete, CASE Werkzeug StP/OMT, welches die in Abschnitt 2.2 dargestellte objektorientierte Softwareentwicklungsmethode OMT implementiert. Im Rahmen der grafischen Spezifikation des dynamischen OMT Modells bietet StP/OMT eine werkzeugspezifische Notation zur grafischen Modellierung von dynamischem Systemverhalten. Bei dieser Notation handelt es sich um eine Statechart [Harel 87] ähnliche Notation zur Darstellung von Zuständen und Zustandsübergängen im Rahmen der Prozeßbeschreibung von Objektklassen eines ereignisgesteuerten Systems [Harel, Pnueli 85]. Diese in Kapitel 4 vorgestellten Zustandsautomaten dienen dem Anwendungsentwickler im Rahmen der Modellierung des Systemverhaltens als eine Ergänzung zu Spezifikationsmodellen des statischen Systemaufbaus wie z.B. *Entity Relationship* Modelle [Chen 76] oder Objektdiagramme.

Die persistente Ablage der StP/OMT Modellierungsobjekte der grafischen Repräsentation der Prozeßspezifikation eines Systems erfolgt in der Entwicklungsdatenbank der Software through Pictures Umgebung, dem StP Repository. Vor der Speicherung der Modellierungsdaten in der Entwicklungsdatenbank erfolgt eine Auswertung dieser Daten, um Konsistenzverletzungen oder Verstöße gegen die Diagrammsyntax zu erkennen und dem Entwickler anzuzeigen. Innerhalb des StP/OMT Entwicklungswerkzeugs ist jedoch keine Funktionalität vorgesehen, die zum Zweck der Entwurfsvalidierung das dynamische StP/OMT Modell simuliert oder animiert und die Analyseergebnisse für eine weitere Verarbeitung sammelt, protokolliert und ausgibt.

Um flexibel, problemadäquat und zuverlässig auf die hohe Anzahl der vom CASE Werkzeug StP/OMT angelegten persistenten Modellierungsobjekte des dynamischen StP/OMT Modells im StP Repository zugreifen zu können, ist eine geeignete Klasse von Softwaresystemen erforderlich. Zur Entwicklung solcher Systeme, der sogenannten persistenten Objektsysteme [Matthes 93], bietet das Tycoon System als integrierte Programmierumgebung einen geeigneten sprachlichen und architektonischen Rahmen zur Implementierung einer in der Abbildung 1.2 dargestellten Umgebung zur Unterstützung des Softwareentwicklungsprozesses durch methodenbasierte Modellierung, Verifikation und Validierung dynamischen Systemverhaltens.

Beim Tycoon System handelt es sich um eine Datenbankprogrammierumgebung, die den sprachlichen und architektonischen Rahmen für eine Definition und Integration generischer Dienste in einer offenen Systemumgebung bereitstellt. Zur Einbindung externer Dienste, z.B. der portablen C++ Klassenbibliothek StarView zur Entwicklung und Gestaltung von grafischen Benutzeroberflächen, wird die algorithmisch vollständige, strikt typisierte und polymorphe Programmiersprache höherer Ordnung TL[3] verwendet. Sie ermöglicht eine Benennung, Bindung und Typisierung der für datenintensive Anwendungen relevanten Objekte und Dienste [Matthes 93].

[3]TL: Tycoon Language

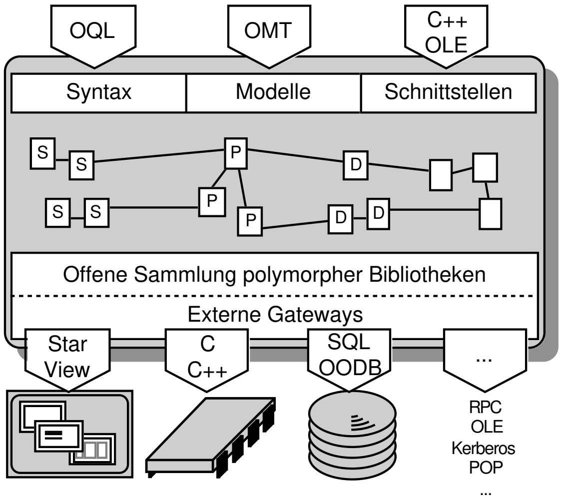


Abbildung 1.3: Integration, Erweiterung und Nutzung generischer Dienste in Tycoon

Wie in Abbildung 1.3 dargestellt, wird der gesamte Prozeß der Integration, Erweiterung, Spezialisierung, Nutzung und Definition generischer Dienste in einem unifizierenden sprachlichen Rahmen realisiert.

Die im Tycoon System verwendete Programmiersprache TL bildet die Schnittstelle zwischen dem Anwendungsprogrammierer und dem Tycoon System und eignet sich wie C sowohl zur Anwendungs- als auch zur Systemprogrammierung. Desweiteren können aufgrund der interaktiven Programmierumgebung des Tycoon Systems mit Hilfe von TL ad hoc Datenbankanfragen gestellt werden. Zusammen mit dem Konzept der Persistenz ist ein inkrementelles Arbeiten über mehrere Benutzersitzungen hinweg möglich. Gleichzeitig unterstützt das Bibliothekskonzept von TL die kontrollierte, gemeinsame Nutzung von Daten und Programmen durch mehrere Benutzer.

TL ermöglicht die Programmierung in verschiedenen Modellierungsstilen. Funktionale und imperative Programmierung werden direkt unterstützt. Aufgrund weitgehend sprachlicher Neutralität wird auch eine Variante des objektorientierten Programmierens ermöglicht. Ein Überblick über die Einflüsse sonstiger Programmiersprachen in TL ist in [Müßig 94] wiedergegeben.

Da bei der Erläuterung der Implementierungskonzepte und -ideen der TL Bibliothek zur Modellierung und Validierung dynamischen Systemverhaltens in den Kapiteln 3, 4 und 5 nicht auf die detaillierte Syntax der Sprache TL eingegangen werden soll, sind Grundkenntnisse in der Tycoon Programmierung notwendig. Eine Einführung in die Tycoon Programmierung ist in [Matthes, Müßig 93] zu finden. Die Anbindung externer Bibliotheken an das Tycoon System wird im Rahmen der Anbindung des Tycoon Systems an die StP Umgebung in Kapitel 3 vorgestellt. Eine detaillierte Darstellung der Anbindung externer Dienste ist in [Mathiske et al. 93] enthalten.

1.3 Zielsetzung

Ziel dieser Arbeit ist die Darstellung einer Umgebung zur Modellierung und Validierung dynamischen Systemverhaltens, die folgende Komponenten umfaßt:

- Das methodenbasierte CASE Werkzeug StP/OMT dient zur grafischen Entwicklung von Verhaltensspezifikationen ereignisgesteuerter Systeme innerhalb des StP/OMT dynamischen Modells. Desweiteren wird es zur Erstellung von Ereignisfluß- und Ereignispfaddiagrammen im Rahmen der Ereignisidentifikation typischer Interaktionssequenzen zwischen der Systemumgebung und dem zu erstellenden Softwaresystem verwendet.
- Die persistente Ablage der StP Diagrammobjekte eines spezifizierten dynamischen Modells erfolgt im StP Repository.
- Das Tycoon System dient als Implementierungstechnologie für die isolierte textuelle Modellierung und Validierung dynamischen Systemverhaltens ereignisgesteuerter Systeme mit TL. Darüberhinaus realisiert es den Zugriff auf die StP Modellierungsobjekte im StP Repository und das modellbasierte *Prototyping* auf der Basis eines dynamischen Modells inklusive der Anbindung externer Dienste.
- StP/OMT wird als Visualisierungswerkzeug für innerhalb des Tycoon Systems entworfene und im StP Repository abgelegte Verhaltensspezifikationen zu entwerfender Systeme verwendet.

Solch eine Modellierungs- und Validierungsumgebung soll exemplarisch aufgebaut und mit Hilfe des Tycoon Systems und seiner Programmiersprache TL implementiert werden.

Desweiteren wird erläutert, inwieweit die Eigenschaften des Tycoon Systems die Implementierung der Modellierungs- und Validierungsumgebung unterstützen und an welchen Stellen der Implementierung konkret auf sie zurückgegriffen wird. Im einzelnen handelt es sich um die folgenden Tycoon Eigenschaften [Matthes 93], die an den entsprechenden Stellen in dieser Arbeit näher erläutert werden:

- Offenheit zur Integration externer Dienste
- Typisierung und Verwendung von Funktionen höherer Ordnung in TL
- Nebenläufigkeit
- Reflektion
- Persistenz

1.4 Struktur der Arbeit

Dieser Einleitung folgt in Kapitel 2 ein Überblick über werkzeugorientierte Systementwicklung. Es werden die Grundzüge und der Nutzen computerunterstützter Anwendungsentwicklung dargestellt. Desweiteren wird objektorientiertes Modellieren und Entwerfen mit OMT, der dem Anwendungsentwicklungsprozeß in dieser Arbeit zugrundeliegenden Entwicklungsmethode, vorgestellt, wobei hier der Schwerpunkt in der OMT methodenbasierten Gestaltung der Analysephase des Softwarelebenszyklus liegt. Im Rahmen dieser Untersuchung ist im Zusammenhang mit der dargestellten Tycoon Modellierungs- und Validierungsumgebung ausschließlich das dynamische OMT Modell Gegenstand der Betrachtung. Eine Erläuterung des OMT Vorgehens zur Ereignisidentifikation typischer Interaktionssequenzen zwischen der Systemumgebung und dem zu entwerfenden Softwaresystem anhand von OMT Ereignisfluß- und Ereignispfaddiagrammen erfolgt in Abschnitt 2.2.2, da es eine Grundlage für die in Abschnitt 5.3.2 vorgeschlagene Vorgehensweise zur modellbasierten Verhaltensvalidierung eines spezifizierten Systems darstellt. Implementiert wird die Methode OMT durch das CASE Werkzeug StP/OMT. Grundlagen der StP Umgebung wie Architekturmerkmale, Basisdienste des StP Kerns und das StP Repository werden dargestellt, da sie für die in Kapitel 3 vorgestellte Anbindung des Tycoon Systems an die StP Umgebung von Bedeutung sind.

Kapitel 3 stellt Grundzüge der Anbindung des Tycoon Systems an die StP Umgebung dar. Neben der Untersuchung der Basisfunktionalität der StP Umgebung wird gezeigt, wie die Abbildung der StP Object Management System Funktionalität auf die Tycoon Ebene vorzunehmen ist und wie die StP Funktionalität in das Tycoon System integriert wird. Mit Hilfe der in Abschnitt 3.3 vorgestellten Tycoon StP Umgebung *StPEnv* wird die gesamte Funktionalität des StP Application Program Interface (API) dem Tycoon Anwendungsentwickler im unifizierenden sprachlichen Rahmen des Tycoon Systems zur Verfügung gestellt. Erst mit Hilfe der Tycoon StP Umgebung *StPEnv* kann das in Abschnitt 5.2.1 vorgestellte modellbasierte *Prototyping* auf der Basis einer grafischen Repräsentation eines StP/OMT dynamischen Modells innerhalb des Tycoon Systems realisiert werden. Allerdings bestehen systembedingte Grenzen an der Schnittstelle zwischen dem StP Application Program Interface und dem Tycoon System, die einen Einfluß auf die Benutzung des externen StP Dienstes aus dem Tycoon System heraus haben. Diese Einschränkungen der Tycoon StP Funktionalität werden dargestellt.

In Kapitel 4 wird die Spezifikationssprache der Statecharts [Harel 87] zur grafischen Modellierung des Verhaltens ereignisgesteuerter Systeme vorgestellt und auf wesentliche, im Rahmen dieser Arbeit exemplarisch betrachtete Statechartelemente, eingeschränkt. Ausgehend von der Statechart ähnlichen Diagrammnotation des dynamischen StP/OMT Modells wird eine Tycoon Modellierungsumgebung zur Entwicklung textueller Tycoon Statecharts, also Statecharts in einer adäquaten TL Repräsentation, vorgestellt. Dabei werden Statechartelemente wie Zustände, Zustandsverfeinerungen und Zustandsübergänge grafisch in einer werkzeugunabhängigen Notation [Harel 87] und in einer daraus resultierenden Tycoon Repräsentation dargestellt. Desweiteren erfolgt eine Erläuterung der den Tycoon Statecharts zugrundeliegenden Semantik von Ausführungsschritten mit dem Ziel, im Rahmen der StP modellbasierten Verhaltensvalidierung einen in einer Tycoon Repräsentation vorliegenden Statechart interaktiv oder auf der Basis eines Ereignispfads auszuführen.

Die Validierung dynamischen Systemverhaltens auf der Basis einer in einer Tycoon Repräsentation eines Statecharts vorliegenden Spezifikation des Verhaltens eines Systems wird in Kapitel 5 beschrieben. Dabei wird die Validierung in den Softwareentwicklungsprozeß auf der Grundlage der betrachteten Entwicklungsmethode OMT eingeordnet. Validierungstechniken, wie die Generierung modellbasierter und validierbarer Prototypen unter Anbindung externer Dienste, werden vorgestellt. Ein exemplarisches Vorgehen zur Validierung des Verhaltens eines Systems wird anhand der Verhaltensspezifikation einer Fernsehersteuerung gezeigt. Außerdem wird dargestellt, wie die Umgebung zur Modellierung und Validierung dynamischen Systemverhaltens in das Tycoon System integriert wird.

Diese Arbeit findet ihren Abschluß in Kapitel 6 mit einer Bewertung, inwieweit sich das Tycoon System als Basistechnologie zur Implementierung der vorgestellten Modellierungs- und Validierungsumgebung eignet und an welchen Stellen der Implementierung auf die in Abschnitt 1.3 dargestellten Eigenschaften des Tycoon Systems zugegriffen wird. Es folgen eine Zusammenfassung und ein Ausblick auf eventuell noch zu berücksichtigende Aspekte im Zusammenhang mit dieser Arbeit.

2. Werkzeugorientierte Systementwicklung

Wie in Kapitel 1 erläutert, soll der Softwareentwicklungsprozeß werkzeugorientiert durchgeführt werden, um die in Abschnitt 1.1 dargestellten Anforderungen an die Systementwicklung erfüllen zu können. Abschnitt 2.1 erläutert diesen werkzeugorientierten Softwareentwicklungsprozeß.

Grundlage einer werkzeugorientierten und computerbasierten Systementwicklung ist in dieser Arbeit die Anwendungsentwicklungsmethode *Object Modeling Technique* OMT [Rumbaugh et al. 91], wobei der Schwerpunkt auf der dynamischen Systemmodellierung liegt (vgl. Abschnitt 2.2). Implementiert wird OMT mit dem in Abschnitt 2.3 erläuterten CASE Werkzeug StP/OMT, welches ein Bestandteil der CASE Umgebung Software through Pictures ist.

2.1 Computerunterstützte Anwendungsentwicklung

Werkzeuge des *Computer Aided Software Engineering* (CASE) sollen neben den in Kapitel 1 erwähnten Verbesserungen zu einer Optimierung des Prozesses der Softwareentwicklung führen, um somit unter anderem die Transparenz und Kommunikation innerhalb der Produktentwicklung zu erhöhen. Dabei hat sich die Bedeutung des Begriffs der computerunterstützten Softwareentwicklung im Verlauf der vergangenen Jahre verändert. Verstand man früher unter automatisierter Unterstützung isolierte Softwarewerkzeuge, so ist es heute eine integrierte Anzahl von Werkzeugen, mit denen alle Phasen des Softwarelebenszyklus sowohl unterstützt als auch umgesetzt werden können [Balzert 93].

2.1.1 Anforderungen an CASE

Grundlage einer automatisierten Anwendungsentwicklung ist nicht allein die Leistungsfähigkeit der einzelnen CASE Werkzeuge. Vielmehr müssen die eingesetzten Werkzeuge im Kontext mit der ihnen zugrundeliegenden CASE Umgebung betrachtet werden. Dabei lassen sich die Anforderungen je nach Kategorie der eingesetzten Computerunterstützung trennen: nach allgemeinen Anforderungen, die an eine CASE Umgebung gestellt werden (Basisanforderungen), und solchen, die von den einzelnen Werkzeugen dieser Umgebung zu erfüllen sind. In diesem Zusammenhang werden in dieser Arbeit sowohl Basisanforderungen an die gesamte CASE Umgebung als auch Anforderungen an die einzelnen CASE Werkzeuge betrachtet.

Basisanforderungen an die CASE Umgebung

- Integrierte Datenhaltung aller Datenbasen jedes separaten CASE Werkzeugs zur Bildung von die Entwicklungsphasen übergreifenden Werkzeugketten
- Erweiterbarkeit um weitere Werkzeuge
- Offenheit und Transparenz bezüglich der Import- und Exportschnittstellen
- Multi- und Interprojektfähigkeit
- Unterstützung verschiedener Entwicklungsplattformen

Anforderungen an einzelne CASE Werkzeuge

- Import- und Exportschnittstellen zur Kommunikation mit fremden Werkzeugen oder der CASE Umgebung
- Integrierbarkeit in die CASE Umgebung

Neben diesen technisch bedingten Anforderungen müssen die eingesetzten Werkzeuge natürlich auf die dem Entwicklungsprozeß zugrundeliegende Methode der Anwendungsentwicklung abgestimmt sein, um überhaupt eine durchgängige Werkzeugkette über alle Phasen des Softwareerstellungsprozesses gewährleisten zu können.

2.1.2 Werkzeugintegration

Eine Unterstützung des Anwendungsentwicklungsprozesses erfolgt, indem sämtliche verwendeten separaten Werkzeuge in einer integralen Art miteinander verknüpft werden, so daß ihre Kombination ein einziges, großes Gesamtwerkzeug, nämlich die komplette Entwicklungsumgebung, ergibt. Die Integrationsbeziehungen zwischen den einzelnen Werkzeugen beziehen sich auf die Präsentation der Mensch-Werkzeug-Schnittstelle (*Presentation Integration*), die von allen Werkzeugen verwendeten Daten (*Data Integration*), den Kontrollfluß zwischen den Werkzeugen innerhalb des Entwicklungsprozesses (*Control Integration*) und den Prozeß des Einsetzens der Werkzeuge selbst (*Process Integration*) [Thomas, Nejmeh 92].

Die Integration aller Werkzeuge einer verteilten Anwendungsentwicklungsumgebung findet ihre Berücksichtigung in der Plattformintegration (*Platform Integration*) [Wassermann 89].

Die Integrationsmechanismen gelten für zwei Klassen von Anwendungsentwicklungswerkzeugen: Werkzeuge zur Unterstützung des Entwicklungsprozesses in den jeweiligen Phasen des Anwendungserstellungsprozesses (*Vertical Tools*) und Werkzeuge zur phasenübergreifenden Unterstützung (*Horizontal Tools*). Phasenübergreifende Entwicklungswerkzeuge umfassen Werkzeuge für Dokumentationszwecke, Projekt- und Prozeßmanagement, Konfigurationsmanagement und das Nachweisen der Berücksichtigung von gestellten Systemanforderungen. Bei Werkzeugen der jeweiligen Projektphasen werden *Upper CASE* Werkzeuge von Werkzeugen des *Lower CASE* unterschieden. *Upper CASE* Werkzeuge werden in den Phasen Analyse und Entwurf eingesetzt, *Lower CASE* Werkzeuge hingegen in den Phasen der Kodierung und des Testens von Anwendungssystemen.

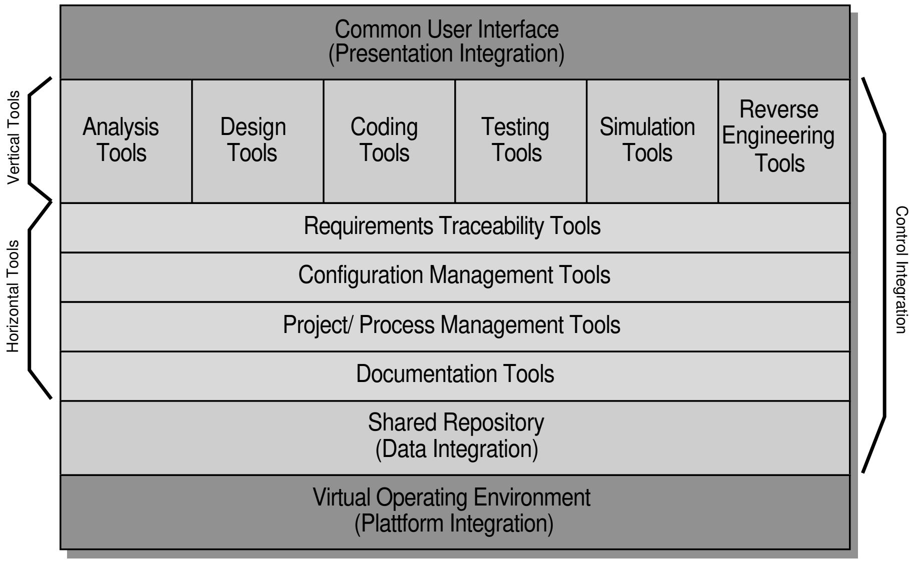


Abbildung 2.1: Integrierte CASE Umgebung

In der Abbildung 2.1 sind die erwähnten Integrationsbeziehungen zwischen Werkzeugen einer computerunterstützten, verteilten Entwicklungsumgebung dargestellt [IDEStPPC 93]. Wichtig ist hierbei, daß die Integration aller Werkzeuge im Rahmen der Datenintegration über das Repository erfolgt. Auf die Entwicklungsdatenbank greifen sowohl die den Entwicklungsprozeß in den jeweiligen Phasen unterstützenden Werkzeuge (*Vertical Tools*) als auch die phasenübergreifenden Werkzeuge (*Horizontal Tools*) zu.

Das in Abschnitt 2.3 dargestellte CASE Werkzeug StP/OMT unterstützt phasenorientiert den Anwendungsentwicklungsprozeß als computerbasiertes Analyse-, Entwurfs- und Kodierungswerkzeug (*Vertical Tool*, vgl. Abbildung 2.1). Außerdem stellt es phasenübergreifende

Funktionalitäten (*Horizontal Tools*) dem Anwendungsentwickler zur Verfügung (z.B. einen Druckmanager zur Aufbereitung und anschließendem Ausdruck der in den einzelnen Phasen erstellten Dokumentationen der separaten StP/OMT Diagramm- und Tabelleneditoren). Die Integration von StP/OMT in die StP Umgebung erfolgt über das StP Repository (*Data Integration*).

2.2 Objektorientiertes Modellieren und Entwerfen mit OMT

Die objektorientierte Methode OMT für die Entwicklung von Anwendungssystemen besteht darin, ein Modell einer Anwendungsdomäne zu erstellen und diesem danach, beim Entwurf des Systems, Implementierungsdetails hinzuzufügen [Rumbaugh et al. 91]. Dabei umfaßt die OMT Methode die folgenden Phasen:

1. In der Analysephase wird ausgehend von einer informellen Problembeschreibung ein Analysemodell erstellt, welches darstellt, was das zu entwerfende System leisten soll. Die Objekte des Modells sind implementierungsunabhängige Abstraktionen der Objekte aus der Anwendungsdomäne. Das Analysemodell enthält keine Implementierungsdetails und richtet sich sowohl an die Anwender als auch an die Systementwickler. Das OMT Analysemodell besteht aus Objektmodell, dynamischem Modell und funktionalem Modell. Abschnitt 2.2.2 erläutert das in dieser Arbeit im Mittelpunkt stehende dynamische Modell näher.

2. Beim Systementwurf wird eine Strategie für die Implementierung des Systems entwickelt. Es werden Entscheidungen über die Gesamtarchitektur des zu erstellenden Anwendungssystems getroffen. Ausgehend vom Analysemodell und einer vorgeschlagenen Architektur wird das System in Teilsysteme organisiert.

3. Innerhalb des Objektentwurfs wird auf Grundlage des Analysemodells ein Entwurfsmodell erstellt, welches Implementierungsdetails enthält. Inhalt des Objektentwurfs sind Datenstrukturen (z.B. Klassen und Assoziationen) und Algorithmen (Methoden), die zur Implementierung jeder Klasse benötigt werden. Die Objektklassen aus der Analysephase werden um diese Datenstrukturen und Algorithmen erweitert. Sowohl anwendungsspezifische als auch implementationsspezifische Objekte werden mit den gleichen objektorientierten Konzepten und der gleichen Notation beschrieben.

4. In der Implementierungsphase werden die beim Objektentwurf entwickelten Objektklassen und Relationen in eine bestimmte Programmiersprache, Datenbank oder Hardwareimplementierung übersetzt.

2.2.1 OMT Metamodell

Innerhalb der Analysephase verwendet die Methode OMT drei unterschiedliche Modelle zur Beschreibung eines Systems. Jedes dieser Modelle (Objektmodell, dynamisches Modell, funktionales Modell) spezifiziert eine eigene Sicht auf das zu entwerfende System. Erst die Integration aller dieser Modelle führt zu einer kompletten Beschreibung eines Systems im OMT Analysemodell [Losavio et al. 94].

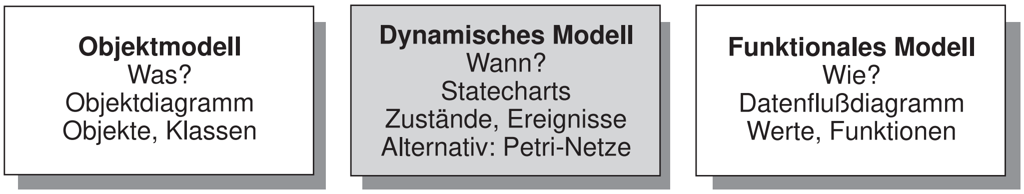


Abbildung 2.2: OMT Modelle

In der Abbildung 2.2 sind die drei Modelle des OMT Analysemodells und ihre Inhalte dargestellt. Das Objektmodell beschreibt die statische Struktur der Objekte in einem System und ihre Relationen zueinander. Es enthält Objektdiagramme, welche in der Form eines Graphen dargestellt werden. Knoten innerhalb eines Objektdiagramms repräsentieren Objektklassen, Kanten die Relationen zwischen diesen Klassen. Das Objektmodell wird vor dem dynamischen und dem funktionalen Modell entwickelt.

Das dynamische OMT Modell beschreibt die Aspekte eines Systems, die sich im Zeitablauf verändern. Das dynamische Modell wird für die Spezifikation der Steuerungsoperationen eines Systems verwendet. Es enthält Zustandsdiagramme, welche als Graph dargestellt werden (vgl. Abschnitt 2.2.2). Knoten innerhalb eines Zustandsdiagramms repräsentieren Systemzustände, Kanten die Transitionen zwischen diesen Zuständen. Spezifiziert wird damit das Systemverhalten im Zeitablauf, welches durch das Eintreten von Ereignissen aus der Systemumgebung bestimmt wird.

Grundlage der Zustandsdiagramme innerhalb des dynamischen OMT Modells sind Statecharts [Harel 87], wobei die innerhalb des dynamischen OMT Modells verwendeten Zustandsdiagramme eine Spezialisierung und Einschränkung der in [Harel 87] dargestellten Statechartnotation darstellen. Alternativ könnten zur Spezifikation dynamischen Systemverhaltens Petri-Netze eingesetzt werden. Eine Abbildung von Statecharts auf Petri-Netze wird in [Fricke 95] vorgestellt.

Das funktionale OMT Modell beschreibt die Transformation von Datenwerten innerhalb des Systems. Es enthält Datenflußdiagramme, welche als Graph dargestellt werden. Knoten innerhalb eines Datenflußdiagramms repräsentieren Prozesse, Kanten Datenflüsse.

Im Rahmen dieser Arbeit ist im folgenden ausschließlich das dynamische OMT Modell Gegenstand der Betrachtung. Eine Darstellung des OMT Objektmodells und des funktionalen OMT Modells ist in [Rumbaugh et al. 91] zu finden.

2.2.2 Modellierung dynamischer Systemaspekte

Das dynamische Modell spezifiziert das zeitabhängige Verhalten des zu erstellenden Systems und seiner Objekte. Die wichtigsten Konzepte der dynamischen Modellierung sind Ereignisse, die Nachrichten an eine Objektklasse repräsentieren und Zustände, die Werte von Objekten repräsentieren. Eine Vorgehensweise zur Konstruktion eines dynamischen Modells umfaßt die folgenden, in diesem Abschnitt betrachteten Schritte:

- Für jedes Objekt des Systems werden die sichtbaren Ereignisse und Reaktionen identifiziert. Diese Ereignisse, die von einer Objektklasse zu einer anderen Objektklasse gesendet werden, können in einem Ereignisflußdiagramm grafisch dargestellt werden. Ein Ereignisflußdiagramm faßt Ereignisse zwischen Objektklassen ohne Berücksichtigung ihrer Reihenfolge des Eintretens zusammen. Kanten in Ereignisflußdiagrammen zeigen mögliche Kontrollflüsse zwischen Objektklassen eines Systems.

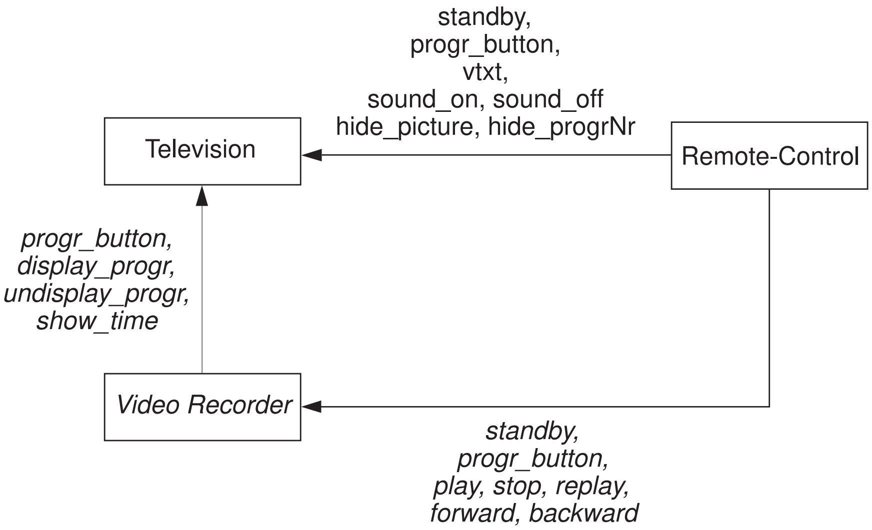


Abbildung 2.3: OMT Ereignisflußdiagramm

In der Abbildung 2.3 ist ein Ereignisflußdiagramm zwischen den in dieser Arbeit betrachteten Objekten einer Fernsehersteuerung dargestellt, wobei die identifizierten Ereignisse zwischen den Objektklassen nur exemplarischen Charakter besitzen. Das spezifizierte Ereignisflußdiagramm hat keinen Anspruch auf Vollständigkeit. Die kursiv dargestellten Ereignisse zwischen der Klasse *Remote-Control* und *Video Recorder* bzw. *Video Recorder* und *Television* werden in den folgenden Abschnitten nicht weiter berücksichtigt.

Von der Objektklasse *Remote-Control* können die in Abbildung 2.3 wiedergegebenen Ereignisse *standby*, *progr_button*, *vtxt*, *sound_on*, *sound_off*, *hide_picture* und *hide_progrNr* an die Klasse *Television* gesendet werden. Bei der Spezifikation des Verhaltens der Klasse *Television* sind diese Ereignisse als Nachrichten zu berücksichtigen.

- Ausgehend von den identifizierten und innerhalb eines Ereignisflußdiagramms dargestellten Ereignissen zwischen den Objektklassen eines Systems werden für anwendungstypische Interaktionssequenzen Szenarios erstellt, um eine zeitlich geordnete Abfolge der Ereignisse zwischen den Objektklassen darzustellen. Ein Szenario bezeichnet in dieser Arbeit eine mögliche zeitliche Abfolge von Ereignissen. Ein Ereignispfad ist eine geordnete Liste tabellarisch angeordneter Ereignisse zwischen Objekten.

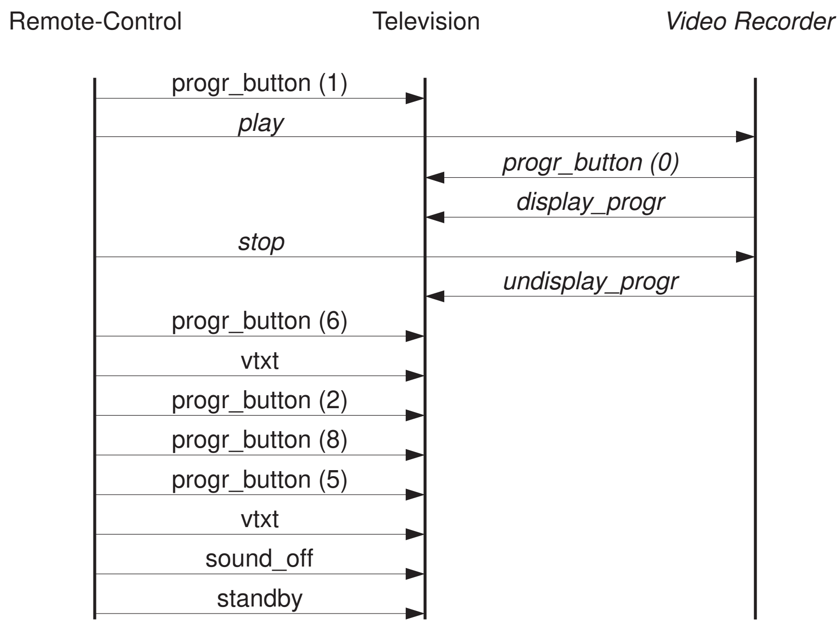


Abbildung 2.4: OMT Ereignispfaddiagramm

In der Abbildung 2.4 wird ein mögliches Szenario zwischen den Klassen *Remote-Control*, *Television* und *Video Recorder* wiedergegeben. Im Rahmen dieser Arbeit werden in den folgenden Abschnitten ausschließlich Ereignisse von der Objektklasse *Remote-Control* an die Klasse *Television* berücksichtigt.

Das in Abbildung 2.4 dargestellte Szenario beginnt mit dem Auslösen des Ereignisses *progr_button* in der Objektklasse *Remote-Control* und endet mit dem Ereignis *standby* an die Klasse *Television*. Spezifiziert ist hier eine Interaktionssequenz, bei der ein Anwender mit Hilfe der Fernbedienung einen Fernseher und einen Videorekorder steuert.

Eingeschaltet wird das Fernsehgerät mit der Programmtaste 1 auf der Fernbedienung (Ereignis *progr_button (1)* von der Klasse *Remote-Control* an die Klasse *Television*). Anschließend wird der Videorekorder gestartet (Ereignis *play*), der den Videokanal am Fernsehgerät einschaltet (Ereignis *progr_button (0)* von der Klasse *Video Recorder* an die Klasse *Television*) und daraufhin das Videobild anzeigt (Ereignis *display_progr*). Danach hält der Benutzer die Wiedergabe des Videorekorders an (Ereignis *stop*), woraufhin das Videobild nicht mehr angezeigt wird (Ereignis *undisplay_program*). Später wechselt der Anwender in das Programm 6 (Ereignis *progr_button (6)*), ruft danach den Videotextmodus auf (Ereignis *vtxt*), wählt die Videotextseite 285 und verläßt den Videotextmodus wieder (Ereignis *vtxt*). Nachdem mit der Fernbedienung der Ton des Fernsehers ausgeschaltet wurde (Ereignis *sound_off*), wird das Fernsehgerät in den Bereitschaftsmodus zurückversetzt (Ereignis *standby*).

Die kursiv dargestellten Ereignisse zwischen den Klassen *Remote-Control* und *Video Recorder* bzw. *Video Recorder* und *Television* werden in den folgenden Abschnitten nicht weiter berücksichtigt.

- Ausgehend von den erstellten Ereignispfaddiagrammen (eine Anzahl von Szenarios) werden die Zustandsdiagramme, die das Verhalten der Objektklassen spezifizieren, erstellt. Grundlage des Verhaltens eines Systems ist das Verhalten seiner Objekte. Für jede Objektklasse wird in der Methode OMT ein Zustandsdiagramm erstellt, das die Ereignisse zeigt, welche die Objektklasse empfängt und sendet. Dabei entspricht jeder Ereignispfad einem Pfad durch das Zustandsdiagramm [Rumbaugh et al. 91].

  Da die Klasse *Remote-Control* in dieser Arbeit nur als Ereignissender auftritt, wird für sie kein Zustandsdiagramm erstellt. Das Zustandsdiagramm für das Verhalten der Klasse *Television* ist in Anhang C.1 in einer an die Darstellung der Zustandsdiagramme in der Methode OMT angelehnten Form wiedergegeben.

Im Zusammenhang mit der Erstellung der Zustandsdiagramme stellt die in Kapitel 5 erläuterte Validierung dynamischen Systemverhaltens die Verbindung zwischen den einzelnen Diagrammen her. Ausgehend von einer Anzahl spezifizierter Ereignispfaddiagramme soll das Verhalten des spezifizierten Systems gegen die Erwartungen des Anwenders untersucht werden. Eine Menge von Ereignispfaddiagrammen gibt die typischen Interaktionssequenzen an, so daß auf deren Basis ein Großteil aller in der Realität eintretenden Ereignisse schon vor dem Entwurf des Systems validiert werden können.

2.3 Die CASE Umgebung Software through Pictures (StP/OMT)

Software through Pictures (StP) der Firma Interactive Development Environments (IDE) entspricht dem in Abschnitt 2.1.2 dargestellten Ansatz integrierter CASE Umgebungen. Bei StP handelt es sich um eine Familie integrierter Mehrbenutzer CASE Umgebungen zur Entwicklung neuer und zum *Re-Engineering* bestehender Systeme.

Jedes StP Produkt beinhaltet eine Familie integrierter Werkzeuge, zum Beispiel, wie in der Abbildung 2.5 skizziert, grafische Editoren oder Tabelleneditoren für die Systemmodellierung, Systeme zur Dokumenterstellung, Schnittstellen zu Versionskontrollsystemen etc.. Die Grafikeditoren innerhalb von StP/OMT unterstützen die in Abschnitt 2.2 vorgestellte objektorientierte Methode OMT zur Softwareentwicklung auf integrale Art.

Alle StP Produkte benutzen eine gemeinsame Entwicklungsdatenbank, das in Abschnitt 2.3.3 erläuterte StP Repository. Somit realisiert die CASE Umgebung Software through Pictures eine in Abschnitt 2.1.2 dargestellte offene Architektur, da die Werkzeugkommunikation über die Entwicklungsdatenbank unter Berücksichtigung von Netzwerk-, Druck- und Benutzeroberflächenstandards erfolgt. Diese offene Architektur ermöglicht eine Erweiterung oder Individualisierung der StP Umgebung. Dies wird in der Arbeit ausgenutzt, um eine Validierungskomponente in die Umgebung zu integrieren (vgl. Kapitel 5).

2.3.1 Architekturmerkmale

Zur Unterstützung der Anwendungsentwicklung bietet StP/OMT, wie in der Abbildung 2.5 angedeutet, eine Menge grafischer Werkzeuge, welche die OMT Entwicklungskonzepte in allen Phasen des Softwarelebenszyklus unterstützen [Klees, Schmauch 94]. Diese die Methode OMT implementierenden grafischen Werkzeuge sind wie allgemeine StP Komponenten oberhalb eines methodenunabhängigen Kerns und unterhalb des StP Desktops angeordnet. Der methodenabhängige Kern besteht aus dem StP Object Management System, Systemdateien (*System Files*) und dem StP Repository.

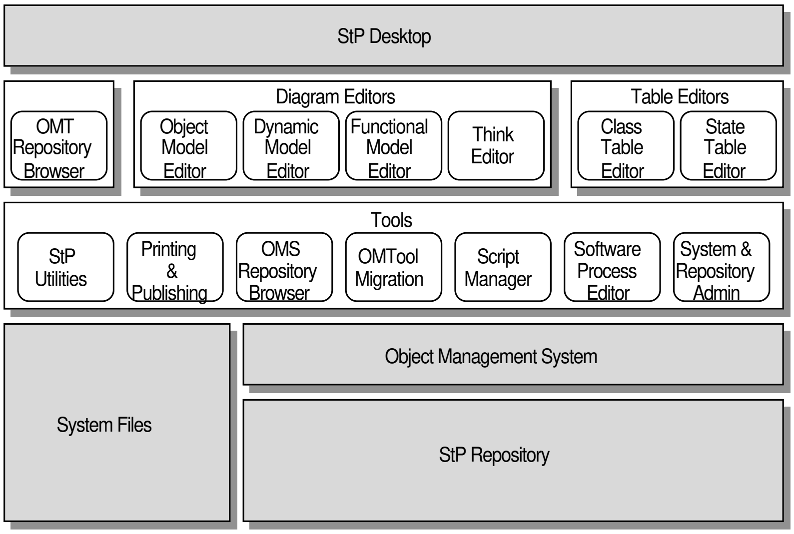


Abbildung 2.5: StP/OMT Architektur

Die Abbildung 2.5 zeigt den schematischen Aufbau einer StP CASE Umgebung am Beispiel von StP/OMT. Die Werkzeuge der oberen Zeile der weiß dargestellten Komponenten werden bei einem anderen CASE Werkzeug der StP Produktfamilie (z.B. StP Information Modeling StP/IM [IDEStPIM 93]) durch methodenadäquate Komponenten substituiert. Die grau unterlegten Grundkomponenten der StP Umgebung stellen die Voraussetzung für die Integration aller StP Werkzeuge dar.

Die in Kapitel 3 vorgestellte Anbindung des Tycoon Systems an die StP Umgebung erfolgt, wie in der Abbildung 3.1 dargestellt, unterhalb des StP Desktops und oberhalb des StP Kerns, also parallel zu den anderen installierten Werkzeugkomponenten.

2.3.2 Basisdienste des StP Kerns

Alle Produkte der StP Familie sind oberhalb eines methodenunabhängigen Kerns angesiedelt. Dieser leistet die in Abschnitt 2.1.1 geforderten grundlegenden Dienste der integrierten StP Entwicklungsumgebung, auf welche die jeweiligen StP CASE Werkzeuge zugreifen. In unterschiedliche Kategorien zusammengefaßt handelt es sich konkret bei diesen Diensten um das Desktopmanagement, die Prozeßmodellierung, die Versionskontrolle und das Konfigurationsmanagement, die Mehrbenutzerkoordination, die Systemindividualisierung und Erweiterbarkeit sowie die optimale Visualisierung komplexer Diagramme [IDEStPTWP 93].

Für die Realisierung dieser Arbeit sind das Objektmanagement, die Repositoryverwaltung und die Gültigkeitsprüfung als Dienste des StP Kerns zu betrachten.

Objektmanagement

Das StP Object Management System (OMS) implementiert das in Abschnitt 2.3.3 dargestellte Informationsmodell StP Persistent Data Model (PDM), nach dem die werkzeugunabhängige Ablage der StP/OMT Modellierungsdaten im Repository erfolgt. Außerdem stellt es den Zugriff auf das StP Repository durch eine Menge von Operationen zur Verfügung. Somit definiert das OMS die Schnittstelle zwischen Werkzeugen der StP Umgebung und dem Repository. Eine Interaktion zwischen einem Werkzeug und dem OMS kann mit Hilfe der in Abschnitt 2.3.5 vorgestellten StP Query and Reporting Language (QRL) oder dem StP Application Program Interface (API) erfolgen.

Repositoryverwaltung

Die Verwaltung des Repositorys erfolgt auf der Grundlage des relationalen Datenbankverwaltungssystems Sybase in einer heterogenen *Client Server* Umgebung. Der Zugriff auf die Datenbankverwaltung wird direkt vom OMS umgesetzt, ohne dabei einen näheren Einblick in das zugrundeliegende relationale Datenbankmodell zu gewähren. Mit Hilfe von Sybase SQL oder ähnlicher StP unabhängiger Datenbankwerkzeuge kann der Zugriff auf die

Repositoryobjekte auch unter Umgehung des StP Object Management Systems realisiert werden. Diese Möglichkeit des Zugriffs auf die Entwicklungsdatenbank ist jedoch kein Untersuchungspunkt dieser Arbeit.

Gültigkeitsprüfung

StP Umgebungen bieten im Zusammenhang mit der Repositoryverwaltung noch einen weiteren Dienst zur Erzielung von Konsistenz innerhalb des Entwicklungsprozesses. Beim Speichern erstellter Modellierungsobjekte im Repository werden diese auf syntaktische Korrektheit überprüft. Sind Integritätsbedingungen im Rahmen des StP Persistent Data Models mißachtet oder ist eine Abbildung eines erweiterten Anwendungsdatentyps auf einen StP PDM Datentyp nicht möglich, so kann das betreffende Objekt nicht im Repository abgelegt werden.

2.3.3 Das StP Repository

Jede einzelne in der Abbildung 2.5 dargestellte StP/OMT Werkzeugkomponente benötigt oder erzeugt eine Vielzahl von Informationen innerhalb des Entwicklungsprozesses. In der verteilten *Client Server* Umgebung entsteht somit innerhalb eines jeden Werkzeugs eine hohe Anzahl von Zugriffsoperationen auf die Datenbasis aller Entwicklungswerkzeuge. Das Repository integriert sämtliche Datenbasen aller Werkzeuge der StP Umgebung. Es ist der „zentrale Speicher“ der Projektinformation [IDEStPOMS 94].

Um eine anwendungsunabhängige Speicherung der Modellierungsobjekte in der Entwicklungsdatenbank zu erreichen und ein hohes Maß an Konsistenz innerhalb des Anwendungsentwicklungsprozesses zu erzielen, verwenden alle Werkzeuge der StP Umgebung das StP Object Management System. Es beinhaltet neben dem in diesem Abschnitt dargestellten Datenmodell und Zugriffsoperationen die in der StP Query and Reporting Language enthaltene OMS Query Language und das StP Application Program Interface (API), um auf Repositoryobjekte werkzeugunabhängig zugreifen zu können.

Persistent Data Model (PDM)

Das StP Persistent Data Model ist ein konzeptuelles Schema, welches die Attribute und Beziehungen von Repositoryobjekten auf der Basis abstrakter Datentypen definiert. Somit entsteht für jede Anwendung die gleiche Zugriffssicht auf die im Repository abgelegten Objekte [IDEStPOMS 94].

Die in der Abbildung 2.6 dargestellten abstrakten PDM Datentypen stehen in einer Subtypbeziehung. Das heißt, daß jeder Typ in einer Sub-/Supertypbeziehung zu einem anderen Typ steht. Der *CASE Type* ist der Supertyp aller anderen Typen dieses Datenmodells. Supertypen, wie der *CASE Type*, der *Annotated Object Type*, der *Reference Type* und der

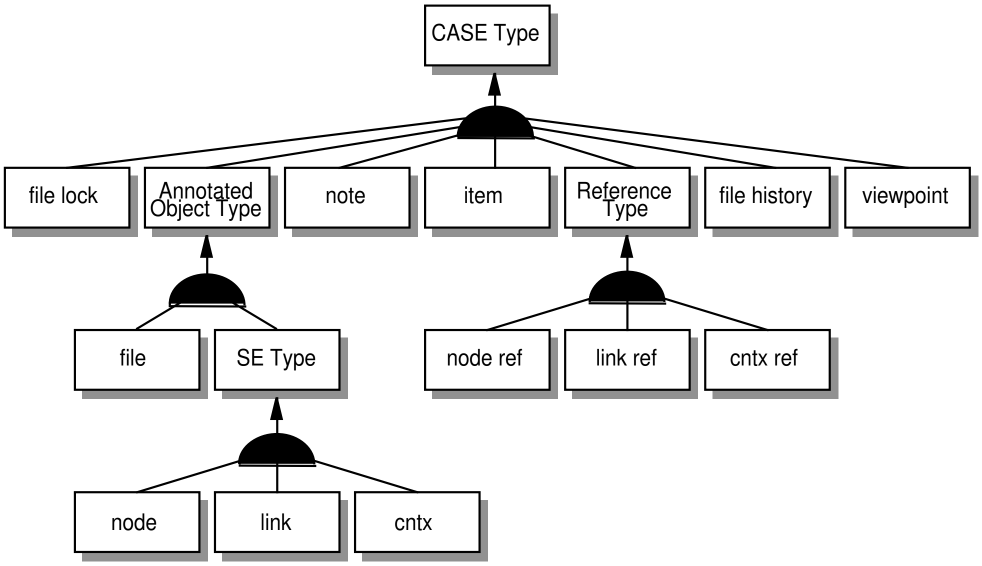


Abbildung 2.6: StP Persistent Data Model (PDM)

Software Engineering (SE) Type werden als abstrakte Klassen bezeichnet und können nicht direkt im Repository gespeichert werden. Nur die Subtypen, die nicht Supertypen anderer Typen sind, können als Repositoryobjekte instantiiert werden. Eine Interpretation sämtlicher StP PDM Datentypen erfolgt an dieser Stelle nicht. Tabellen aller PDM Datentypen und ihrer Attribute befinden sich in [IDEStPOMS 94].

PDM Datentypen und Beziehungen

Abgesehen von den konzeptuellen Sub-/Supertypbeziehungen stehen die StP PDM Datentypen noch in einer für die Ablage der Modellierungsobjekte im StP Repository wesentlichen Beziehung: Zwischen Objekten der PDM Datentypen gelten referentielle Beziehungen [IDEStPOMS 94].

Die in der Abbildung 2.7 dargestellten Integritätsbedingungen zur Speicherung von Objekten im Repository haben einen großen Einfluß auf die in Abschnitt 3.3 vorgeschlagene Modularisierung der zu erstellenden Bibliothek auf der Tycoon Ebene hinsichtlich ihrer Typisierung.

Beispielsweise kann ein *cntx* Objekt nicht im Repository abgelegt werden, solange das referenzierte *link* Objekt dort noch nicht vorhanden ist. In der Abbildung 2.7 stellen Pfeile die referentiellen Abhängigkeiten zwischen Objekten dar. Die Existenz eines *link* Objekts ist die Voraussetzung für die Ablage des abhängigen *cntx* Objekts im Repository.

Objektattribute, deren Name auf *_id* endet, verweisen also auf ein Repositoryobjekt (vgl. Fremdschlüssel im relationalen Datenmodell). Ein mit einem Stern (*) gekennzeichnetes At-

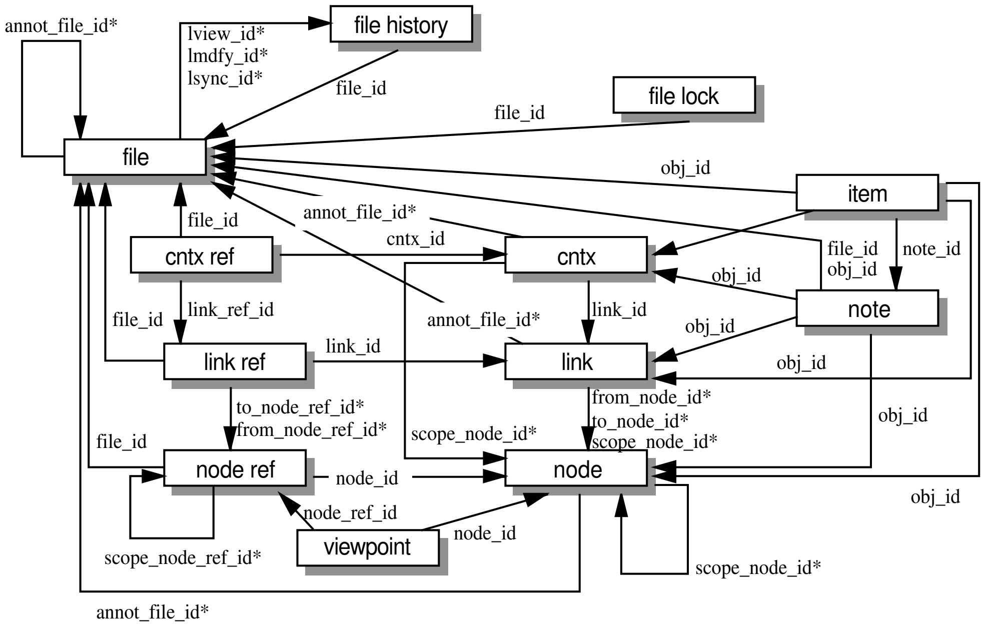


Abbildung 2.7: Integritätsbedingungen zwischen StP PDM Datentypen

tribut eines Repositoryobjekts, z.B. *annot_file_id**, kann den Wert 0 annehmen, was bedeutet, daß das Object Management System bei der Speicherung dieses Objekts im Repository nicht auf der Existenz eines von diesem Objekt referenzierten Objekts besteht.

2.3.4 Anwendungssicht auf das StP Repository

Die Verbindung zwischen den Modellkomponenten von StP/OMT Werkzeugen und den Datentypen des StP Persistent Data Models wird mit Hilfe der Abbildungsdatei `omt.app.types` realisiert. Die Anwendungssicht auf das StP Repository ist in der Abbildung 2.8 dargestellt [IDEStPOMS 94]. Darin werden StP/OMT Werkzeuge als Anwendung bezeichnet.

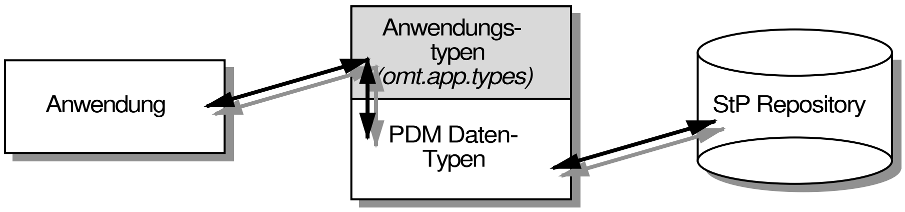


Abbildung 2.8: Anwendungssicht auf das StP Repository

Bei der Abbildungsdatei `omt.app.types` handelt es sich um eine um beliebige Anwendungsdatentypen erweiterbare Datei, welche die von StP Werkzeugen referenzierten Anwendungsdatentypen auf die vom StP Repository verwendeten PDM Datentypen abbildet [IDESt-POMS 94].

In der Abbildung 2.9 ist ein Ausschnitt der erweiterbaren Datei `omt.app.types` wiedergegeben, in dem einige Anwendungsdatentypen des dynamischen StP/OMT Modells definiert sind (*Application Type*), die auf einen StP *Persistent CASE Object Type* (hier: `ItemObject`, `LinkObject`, `NodeObject`) abgebildet werden.

Persistent CASE Object Type	Identifier	Application Type	Attributes
ItemObject	501701	OMTTransitionEvent	{{DataType "String"}}
ItemObject	501702	OMTTransitionAttributes	{{DataType "String"}}
ItemObject	501703	OMTTransitionCondition	{{DataType "String"}}
ItemObject	501704	OMTTransitionAction	{{DataType "String"}}
LinkObject	601544	OMTTransition	{}
LinkObject	601551	OMTStateEntryAction	{}
LinkObject	601552	OMTStateExitAction	{}
LinkObject	601554	OMTStateActivity	{}
NodeObject	701541	OMTState	{{Annot "annotation/OMTState"}}
NodeObject	701542	OMTConcurrentState	{{Annot "annotation/OMTState"}}
NodeObject	701543	OMTStartState	{}
NodeObject	701545	OMTSplitControl	{}
NodeObject	701548	OMTActivity	{}
NodeObject	701549	OMTAction	{}

Abbildung 2.9: Abbildung von StP Anwendungsdatentypen auf PDM Datentypen

Dabei wird jedem Anwendungsdatentypen ein eindeutiger Identifikator (*Identifier*) vom Typ `int` zugeordnet, der im StP Repository mit den entsprechenden Repositoryobjekten abgelegt wird. Jeder *Persistent CASE Object Type* entspricht einem gültigen PDM Datentyp (hier: Typen *item*, *link*, *node*), dessen Attribute an jeden auf diese Weise definierten Anwendungsdatentypen vererbt werden (vgl. Abbildung 2.6 auf Seite 22). Zusätzlich zu den geerbten Eigenschaften kann ein Anwendungsdatentyp um zusätzliche Attribute spezialisiert werden, was mit Hilfe einer Datei (*Annotation File*) geschieht.

Ein konkretes Beispiel für die in der Datei `omt.app.types` spezifizierte Abbildung von StP Anwendungsdatentypen des dynamischen StP/OMT Modells auf PDM Datentypen ist in Abbildung 2.9 dargestellt. Berücksichtigt sind hierbei die Anwendungsdatentypen, die für die Generierung eines Tycoon Validierungsprototypen (vgl. Abschnitt 5.2.1) aus einem dynamischen StP/OMT Modell von Bedeutung sind. Die in der Abbildung 2.9 wiedergegebenen PDM Datentypen repräsentieren die in den Abbildungen 4.1 und 4.3 vorgestellten Statechart Syntaxelemente für die Tycoon Modellierung von dynamischem Systemverhalten.

2.3.5 Zugriffsmöglichkeiten auf StP Repositoryobjekte

Das StP Object Management System stellt dem Anwender neben dem StP Repository Browser die werkzeugunabhängigen Zugriffsmöglichkeiten der StP Query and Reporting Language (QRL) und des StP Application Program Interface (API) zur Verfügung, um direkt auf Repositoryobjekte zuzugreifen. Im Rahmen der StP QRL müssen vom Anwender QRL Skripte erstellt werden, in welchen der Repositoryzugriff mit Hilfe der OMS Query Language realisiert wird. StP QRL Skripte sind als integraler Bestandteil des StP Query and Reporting Systems (QRS) zu betrachten. Darin werden mit Hilfe des StP Query and Reporting Processors (qrp) die in einem StP QRL Skript enthaltenen QRL Befehle im Rahmen der Skriptinterpretation in OMS Anfragen umgesetzt, um so den eigentlichen Zugriff auf die Repositoryobjekte über die Schnittstelle des StP Object Management Systems zu erzielen. Eine Darstellung von StP QRL Skripten und des QRS selbst ist in [IDEStPQRS 94] enthalten.

Neben der Einbettung von OMS Anfragen in Query and Reporting Language Skripte ist es ebenfalls möglich, die Repository Anfragen in Funktionen des StP API in der Form von an die API Funktionen übergebenen Argumenten zu integrieren. Dabei werden die StP API Funktionen in C Programme integriert und zur Laufzeit des C Programms ausgeführt. Das StP Application Program Interface bildet somit eine Schnittstelle zwischen C Anwendungsprogrammen und dem StP Repository. Im Rahmen der Anbindung des Tycoon Systems an die StP Umgebung (vgl. Kapitel 3) werden die C Funktionen des StP API für den Repositoryzugriff eingesetzt. In der Tabelle 2.1 sind exemplarisch einige ausgewählte StP API Funktionen wiedergegeben. Weitere Funktionen sind in [IDEStPOMS 94] aufgeführt.

Eingesetzt wird die Funktionalität des StP API im Rahmen der in Abschnitt 5.2.1 vorgestellten Generierung eines auf einem dynamischen StP/OMT Modell basierenden Tycoon Prototypen zur Validierung der Verhaltensspezifikation eines zu entwerfenden Systems. Dabei werden z.B. OMS Query and Reporting Language Anweisungen als Parameter der StP API Funktion `oms_<pdm_typ>_find_by_query` übergeben. Ziel ist, die im StP Repository abgelegten Objekte eines dynamischen StP/OMT Modells zu identifizieren und auf sie zuzugreifen, um sie anschließend auf der Tycoon Ebene in eine adäquate Form eines Tycoon Statecharts zu überführen.

Eine exemplarische StP QRL Anweisung zum Auffinden der *node* Objekte mit dem Identifikator `701541`, der einen mit dem Diagrammeditor des dynamischen StP/OMT Modells erstellten Zustand repräsentiert lautet:

```
for_each_in_select("node[id = 701541]", aktuellerNode)
```

Bei dieser OMS Anfrage wird der erste Wert des Anfrageergebnisses an die C Variable `aktuellerNode` übergeben. Sämtliche StP API Funktionen werden, wie in Kapitel 3 dargestellt, auf Tycoon Funktionen übertragen, um aus dem Tycoon System heraus die gesamte Funktionalität des StP Object Management Systems benutzen zu können.

Funktion	**Beschreibung**
oms_<pdm_typ>_create	Erzeugt ein neues Repositoryobjekt im Hauptspeicher und gibt einen darauf weisenden Zeiger zurück.
oms_<pdm_typ>_find	Sucht im aktuellen Repository nach dem mit den Funktionsparametern spezifizierten Objekt. Dieses wird in eine neue oms_<pdm_typ>_tp Struktur kopiert. Ein darauf weisender Zeiger wird zurückgegeben. Ansonsten wird der NULL-Zeiger zurückgegeben.
oms_<pdm_typ>_find_by_id	Sucht im aktuellen Repository nach dem mit den Funktionsparametern spezifizierten Objekt. Dieses wird in eine neue oms_<pdm_typ>_tp Struktur kopiert. Ein darauf weisender Zeiger wird zurückgegeben. Ansonsten wird der NULL-Zeiger zurückgegeben.
oms_<pdm_typ>_find_by_query	Sucht im aktuellen Repository nach dem mit den „OMS Anfragebedingungen“ übereinstimmenden Objekt. Dieses wird in eine oms_<pdm_typ>_tp Struktur kopiert. Ein darauf weisender Zeiger wird zurückgegeben. Ansonsten wird der NULL-Zeiger zurückgegeben.
oms_<pdm_typ>_free	Nimmt die Adresse des übergebenen Zeigers auf die oms_<pdm_typ>_tp Struktur und gibt den damit assoziierten Hauptspeicherbereich frei.
oms_<pdm_typ>_print	Nimmt die Adresse des übergebenen Zeigers auf die oms_<pdm_typ>_tp Struktur und gibt alle Attributnamen und deren Werte in einer zu spezifizierenden Datei aus.
oms_<pdm_typ>_ <attribut>_asgn	Weist einem Repositoryobjekt-Attribut einen neuen Wert zu.
oms_<pdm_typ>_update	Aktualisiert oder fügt im Hauptspeicher angelegte Strukturen im StP Repository als Objekte ein. Tritt ein Fehler auf oder sind referentielle Integritätsbedingungen nicht erfüllt, gibt die Funktion OMS_FAIL zurück, ansonsten OMS_SUCCEED.
oms_sys_repos_open	Öffnet das spezifizierte Repository für den Zugriff.
oms_repos_close	Schließt das aktuelle Repository nach dem Abschluß der Zugriffsoperationen.

Tabelle 2.1: Ausgewählte StP Application Program Interface Funktionen

3. Anbindung des Tycoon Systems an die StP Umgebung

Die Anbindung des Tycoon Systems als Basistechnologie für die Implementierung der in Abschnitt 5.4 dargestellten Modellierungs- und Validierungsumgebung an die StP Umgebung erscheint aus Sicht des Anwendungsentwicklers, wie aus der Abbildung 3.1 ersichtlich wird, als Erweiterung der in der Abbildung 2.5 vorgestellten StP/OMT Architektur.

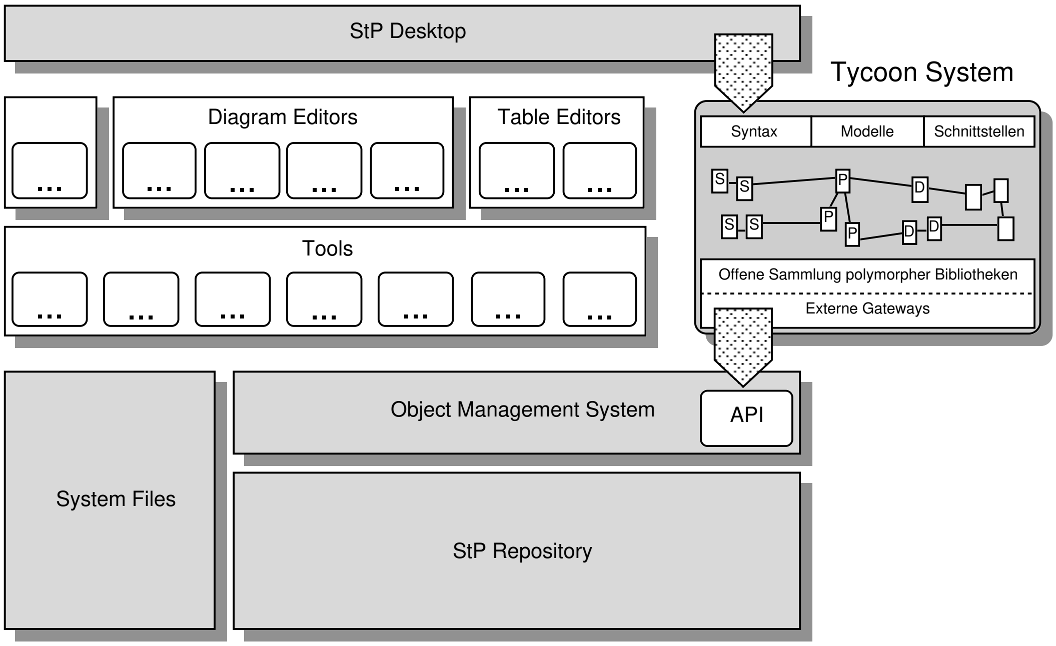


Abbildung 3.1: Anbindung des Tycoon Systems an die StP Umgebung

Dabei verhält sich das Tycoon System als integrierte Entwicklungsumgebung hinsichtlich seines Aufrufs und Zugriffs auf das StP Repository wie sonstige, in die StP Umgebung integrierte StP Werkzeuge, z.B. Diagrammeditoren, Tabelleneditoren etc. [Nietsch 96]. Der

Aufruf des Tycoon Systems erfolgt über ein Piktogramm auf dem StP Desktop. Der Zugriff vom Tycoon System auf die StP Repositoryobjekte erfolgt über die vom StP Application Program Interface zur Verfügung gestellten C Funktionen, welche die in Abschnitt 3.1 dargestellte Basisfunktionalität der StP Umgebung für Anwendungsentwickler verfügbar machen. Aus Sicht des Tycoon Systems stellt das StP Object Management System einen generischen, externen Dienst dar.

Als Resultat dieser Anbindung des Tycoon Systems an die StP Umgebung entsteht im Rahmen des Tycoon Systems die generische Bibliothek *StPEnv*, die dem Programmierer im unifizierenden sprachlichen Raum der Tycoon Umgebung die gesamte, in Abschnitt 3.1 dargestellte, StP Funktionalität für den Repositoryzugriff bietet.

3.1 Basisfunktionalität der StP Umgebung

Die generische Tycoon Bibliothek für den StP Repositoryzugriff muß für die in Abschnitt 5.2.1 dargestellte Generierung eines Tycoon Validierungsprototypen aus einem dynamischen StP/OMT Modell mindestens die folgenden an sie gestellten Anforderungen befriedigen, welche sich an der Funktionalität des StP Application Program Interface orientieren [IDEStPOMS 94]:

- Repositorymanagement
- Initiieren, Überwachen und Abschließen von StP Repositorytransaktionen
- Interpretieren von OMS Anfragen
- Generieren und Auffinden von Repositoryobjekten
- Auffinden, Manipulieren und Ausgeben von Attributwerten der spezifizierten Repositoryobjekte
- Durchführen von Operationen auf zu einer Einheit zusammengefaßten Objekten innerhalb des StP Repositorys im Rahmen der abstrakten Implementationsdatentypen *Collection* und *Big Collection*
- Durchführen der Fehlerbehandlung bei nicht erfolgreich abgeschlossenen Repositoryoperationen

Diese Anforderungen an den Funktionsumfang der StP Tycoon Bibliothek finden ihre Berücksichtigung in der Abbildung der Operationen des StP API auf die entsprechenden Tycoon Funktionen und Module.

3.2 Abbildung der StP OMS Funktionalität auf die Tycoon Ebene

Die gesamte Funktionalität für den Repositoryzugriff wird vom StP Application Program Interface zur Verfügung gestellt, welches eine C basierte Bibliothek von Funktionen und Datentypen darstellt. Auf der Tycoon Ebene sollen beim Zugriff auf Repositoryobjekte im Rahmen der in Abschnitt 3.3 vorgestellten StP Umgebung, der Tycoon Bibliothek *StPEnv*, ausschließlich Funktionen und Datentypen auf der Basis von TL, der Programmiersprache des Tycoon Systems, benutzt werden. Somit kann der Tycoon Programmierer ohne direkte StP API C Funktionsaufrufe auf das StP Repository zugreifen. Diese Vorgehensweise macht es erforderlich, daß innerhalb der Tycoon Umgebung eine geeignete Abbildung der C basierten API Funktionen auf adäquate TL Funktionsdeklarationen erfolgt.

In diesem Abschnitt wird die Abbildung der StP API Funktionalität auf Tycoon Funktionalität exemplarisch am Beispiel einiger Funktionen des StP PDM Datentypen `node` dargestellt (vgl. Abbildungen 2.6 und 2.7). Diese Vorgehensweise ist auch für alle übrigen StP PDM Datentypen und sonstige innerhalb des StP Application Program Interface verwendeten Datentypen und Funktionen gültig.

Außer einer einfachen Abbildung der StP API Funktionalität wird die Mächtigkeit der StP Umgebung auf der Tycoon Ebene durch zusätzliche, den Umfang der eigentlichen API Funktionalität erweiternde Charakteristika ergänzt (vgl. Abschnitt 3.3).

3.2.1 Tycoon C Calls für den StP API Funktionsaufruf

Das Tycoon System bietet einen generischen Mechanismus, um auf in externen Programmiersprachen implementierte Funktionalität zugreifen zu können [Mathiske et al. 93]. Eine externe C Funktion kann mit Hilfe der Tycoon Funktion *bind* an einen Tycoon Bezeichner gebunden werden:

> ***let*** *bspFkt =*
> *bind(:**Fun**(:Int) :Int "bsp_bibliothek.so" "bsp_funktion" "ii")*

Hier wird die in der dynamischen C Bibliothek `bsp_bibliothek.so` vorhandene C Funktion `bsp_funktion` an den Tycoon Bezeichner *bspFkt* statisch gebunden. Der Wert dieses Bezeichners ist vom Typ ***Fun****(:Int) :Int*. Innerhalb der Tycoon Umgebung kann die C Funktion `bsp_funktion` daraufhin wie folgt indirekt aufgerufen werden:

> ***let*** *ergebnis :Int = bspFkt(20)*

Daß sich eine C Funktion hinter der Tycoon Funktion *bspFkt* verbirgt, ist für den Tycoon Programmierer nicht ersichtlich. Diese Vorgehensweise wird im Rahmen des Aufrufs der StP Application Program Interface Funktionen von der Tycoon Ebene aus im weiteren Verlauf dieser Arbeit verfolgt. Weitere Informationen über die Benutzung externer C Bibliotheken aus dem Tycoon System heraus sind in [Mathiske et al. 93] enthalten.

3.2.2 Abbildung von StP API Operationen auf Tycoon Funktionen

Die Abbildung der StP API Funktionen auf Tycoon Funktionen erfolgt unter Verwendung der eben dargestellten Vorgehensweise in fünf Schritten:

1. Auf der Systemebene muß als Grundlage für die Tycoon Funktion *bind* aus den vorhandenen dynamischen StP Bibliotheken eine einzige dynamische Bibliothek erstellt werden, welche die gesamte benötigte StP Funktionalität des StP API zur Verfügung stellt. Der Name dieser Bibliothek wird beim Anbinden einer StP API Funktion der Tycoon Funktion *bind* als Parameter übergeben.

   Begründet wird die Generierung einer einzigen dynamischen Bibliothek dadurch, daß beim Aufruf einer gebundenen API Funktion eventuell weitere Operationen oder Datentypen aus anderen dynamischen StP Bibliotheken referenziert werden. Um auf nicht bekanntgegebenen Funktionen oder Datentypen basierende Fehler zu vermeiden, werden sämtliche dynamischen Bibliotheken miteinander verbunden. Ein entsprechendes Generierungsskript zum Erstellen der Bibliothek `libstp.so` ist in Anhang A.1 enthalten.

2. Eine Voraussetzung für eine erfolgreiche Ausführung der StP API C Funktionen bzw. einen erfolgreichen Entwicklungsdatenbankzugriff ist ein Eintrag `stp_file_path` mit den entsprechenden Pfaden zu den StP Dateien innerhalb der Datei `ToolInfo` [IDEStPCust 94], in welcher grundlegende Initialisierungen der referenzierten StP Systemvariablen und Pfade zu den benötigten Dateien der einzelnen StP Produkte anzugeben sind. Die Datei `ToolInfo` muß für die Benutzung des StP Application Program Interface aus dem Tycoon System heraus um die in Anhang A.2 dargestellte Pfadangabe zu den StP Systemdateien ergänzt werden.

3. Auf der Tycoon Ebene müssen die in Anhang A.3 wiedergegebenen Systemumgebungsvariablen innerhalb jedes neu gestarteten Tycoon Subprozesses gesetzt werden. Nur so kann ein erfolgreicher Zugriff auf Objekte des StP Repositorys vom Tycoon System aus erreicht werden.

   Zu diesem Zweck gibt es in der Tycoon Bibliothek *MachineEnv* das Modul *uxEnvironment*, mit dessen Funktion *set(name, value :String) :**Ok*** die entsprechenden StP Systemvariablen zu Beginn eines Tycoon Subprozesses initialisiert werden können. Innerhalb der StP Tycoon Bibliothek *StPEnv* übernimmt die Funktion *omsRepository.init()* diese Aufgabe.

4. Daraufhin können auf der Tycoon Ebene die C Funktionen an die entsprechenden Bezeichner gebunden und anschließend wie übliche Tycoon Funktionen benutzt werden. Ein Beispiel aus dem Tycoon Modul *omsNode* der Bibliothek *StPEnv* für das Auffinden eines StP Repositoryobjekts des StP PDM Datentyps `node` anhand seines Namensattributs ist in Anhang A.4 wiedergegeben.

5. Der Tycoon Programmierer kann den Zugriff auf das StP Repository nach erfolgter Initialisierung der StP Systemvariablen und einem Import der entsprechenden Module der in Abschnitt 3.3 wiedergegebenen StP Umgebung *StPEnv* über die dort deklarierten Tycoon Funktionen nach dem folgenden Muster durchführen:

   ```
   import omsRepository omsNode

   let projdir = "/local/dbis11/software/StP/project"
   let system = "nietsch"

   omsRepository.init()

   let rep = omsRepository.connectSystem(projdir system)

   let node = omsNode.find(rep "Television"
     omsApplicationType.type omsObjectId.optional() " ")
   ...

   omsRepository.disconnect(rep)
   ```

   Nach Initialisierung der StP Systemvariablen durch die Funktion *omsRepository.init* wird eine Anzahl von Repositoryoperationen stets durch das Öffnen des Repositorys mit Hilfe der TL Funktion *omsRepository.connectSystem* oder *omsRepository.connectRep* und späteres Schließen (*omsRepository.disconnect*) eingerahmt.

   Innerhalb dieses Beispiels wird auf alle StP Repositoryobjekte des spezifizierten StP Projektverzeichnisses (*projdir*) und StP Systems (*system*) zugegriffen. Zurückgegeben werden die Repositoryobjekte, welche vom StP PDM Datentypen `node` sind und deren Namensattribut den Wert *Television* besitzt.

Eine schematische Darstellung eines StP API Funktionsaufrufs für den Zugriff auf das StP Repository aus dem Tycoon System ist in Abbildung 3.2 dargestellt.

Als Resultat des Tycoon Aufrufs der an den Tycoon Bezeichner *omsNodeFind* gebundenen C Funktion `oms_node_find` wird ein Zeiger auf eine Hauptspeicheradresse zurückgegeben, in welcher die C Struktur als Ergebnis der Repositoryanfrage abgelegt ist (hier: `oms_node_tp`).

3.3 Die Tycoon Umgebung StPEnv für den StP Repositoryzugriff

Die Bibliothek *StPEnv* bietet auf der Tycoon Ebene die gesamte Funktionalität des StP Application Program Interface, um indirekt unter Verwendung der Funktionalität des StP

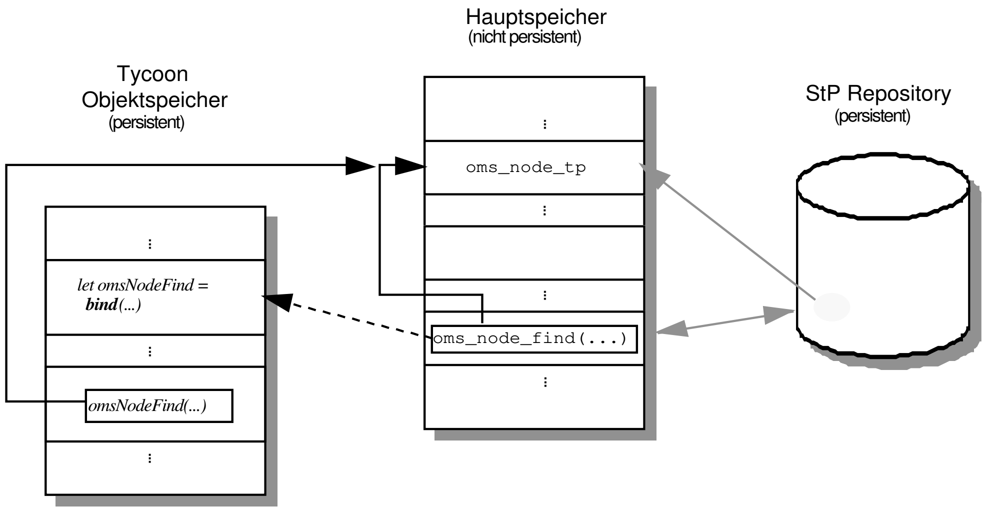


Abbildung 3.2: StP API Funktionsaufruf vom Tycoon System

Object Management Systems auf Repositoryobjekte innerhalb des StP Repositorys zugreifen zu können. Dabei finden die in Abschnitt 3.1 dargestellten Anforderungen an die StP Bibliothek ihre Berücksichtigung in den unterschiedlichen, in Abschnitt 3.3.2 wiedergegebenen Modulen.

3.3.1 Modellierungsaspekte

Die Grundlage der Modellierung der Bibliothek *StPEnv* stellt das in der Abbildung 2.6 auf Seite 22 wiedergegebene StP Persistent Data Model (PDM) dar. Einschränkend gilt, daß die abstrakten Klassen *CASE Type*, *Annotated Object Type*, *Reference Type* und *SE Type* keine Berücksichtigung im Rahmen der Modellierung der StP Bibliothek finden, da nur Objekte ihrer Subtypen, die ihrerseits keine Supertypen anderer StP PDM Datentypen sind, im Repository abgelegt werden können.

Desweiteren basiert die Modularisierung der Bibliothek *StPEnv* auf den in der Abbildung 2.7 auf Seite 23 aufgeführten Beziehungen zwischen den einzelnen Datentypen des StP PDM. Die Art der Ausprägung der Beziehung zwischen zwei Repositoryobjekten ist dabei unterschiedlich zu berücksichtigen. Handelt es sich um eine 1:1 Beziehung, ist dieses durch das Vorhandensein eines betreffenden Attributs des referenzierten Objekts darstellbar. 1:n Beziehungen sind durch die Bildung von „Ansammlungen“ referenzierter Objekte, *Collections* oder *Big Collections*, darzustellen. Außerdem sind die sonstigen *CASE Type* Attribute und ihre Verwaltungs- und Manipulationsfunktionen in angemessener Weise zu berücksichtigen. Eine Erläuterung der *Case Type* Attribute und deren Datentypen ist in [IDEStPOMS 94] enthalten.

3.3.2 Bibliotheksumfang

In der Abbildung 3.3 ist die Modulstruktur der entwickelten Tycoon StP Umgebung *StPEnv* dargestellt. Die Umgebung umfaßt drei verschiedene Arten von Modulen, welche unterschiedliche Funktionalitäten beinhalten.

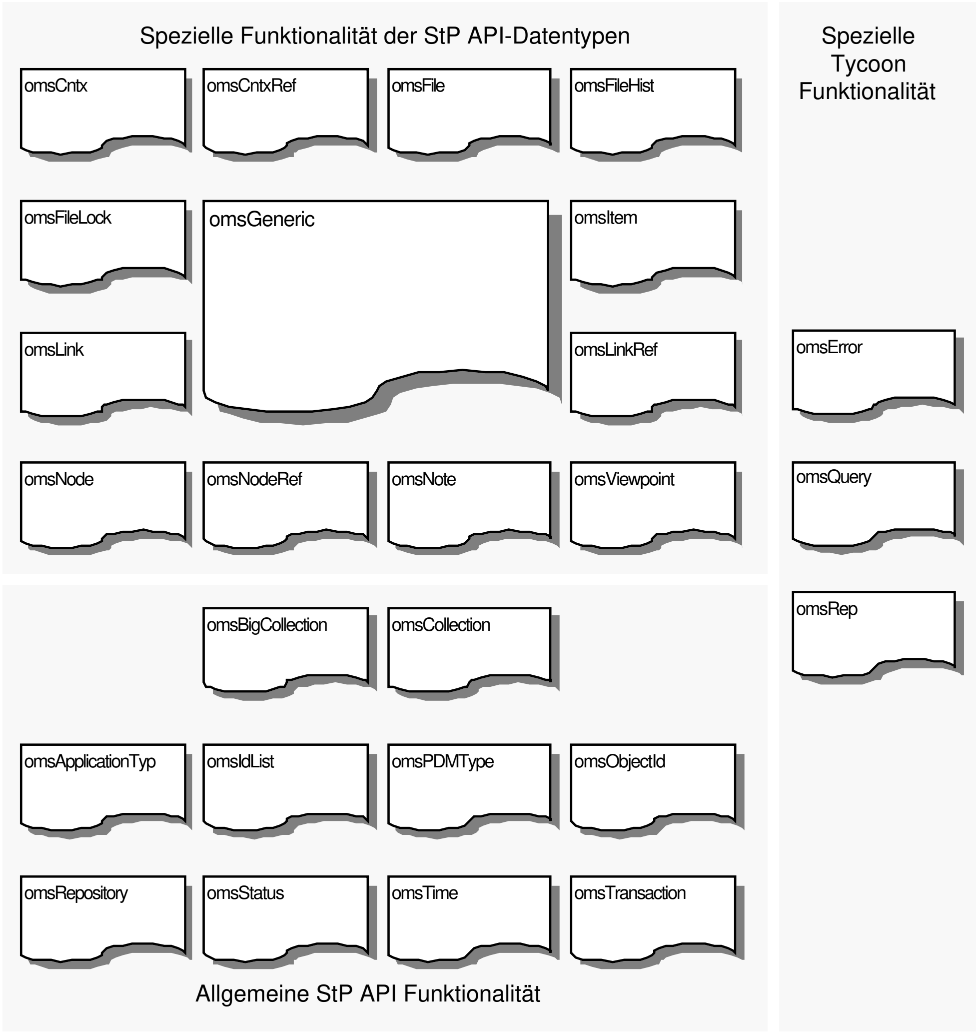


Abbildung 3.3: Tycoon StP Bibliothek *StPEnv*

Spezielle Funktionalität auf den PDM Datentypen

Die Module dieser Kategorie stellen dem Tycoon Programmierer die Funktionen auf den PDM Datentypen zur Verfügung (vgl. Abbildungen 2.6 und 2.7). Die Datentypen werden durch die StP API C Datenstrukturen oms_<pdm_typ>_tp repräsentiert. Sie bilden Operationen auf den C Datenstrukturen `oms_cntx_tp`, `oms_cntx_ref_tp`, `oms_file_tp`, `oms_file_hist_tp`, `oms_file_lock_tp`, `oms_item_tp`, `oms_link_tp`, `oms_link_ref_tp`, `oms_node_tp`, `oms_node_ref_tp`, `oms_note_tp` und `oms_viewpoint_tp` des StP Application Program Interface ab. Beispielsweise finden die Operationen auf dem PDM Datentyp `node`, repräsentiert durch den C Datentyp `oms_node_tp`, ihre Entsprechung auf der Tycoon Ebene im Modul *omsNode*. Das Modul *omsGeneric* umfaßt generische Funktionen aller PDM Datentypen. Es wird ausschließlich in den Modulen dieser Kategorie importiert. Operationen auf Kollektionen (*Collection* oder *Big Collection* Typen) sind in den Modulen der nächsten Kategorie enthalten.

Allgemeine StP API Funktionalität

Innerhalb dieser Kategorie kann zwischen zwei Arten von Modulen unterschieden werden. Die Module *omsBigCollection* und *omsCollection* stellen generische Funktionen zur Verfügung, mit deren Hilfe *Big Collections* oder *Collections* von mehreren Objekten eines zu spezifizierenden StP PDM Datentypen erstellt werden können. Die restlichen Module stellen Funktionen auf den StP API C Datenstrukturen `oms_app_type_tp`, `oms_idlist_tp`, `oms_pdm_type_tp`, `oms_object_id_tp`, `oms_repos_tp`, `oms_status_tp`, `oms_time_tp` und `oms_txn_tp` bereit, die im Rahmen dieser Arbeit nicht vertiefend untersucht werden.

Spezielle Tycoon Funktionalität

Module dieser Kategorie erweitern die StP Funktionalität um eine im Rahmen des StP API nicht vorhandene Funktionalität, welche auf der Tycoon Ebene benötigt wird, um die Realisierung der Repositoryanbindung überhaupt erstellen zu können.

Eine nähere Beschreibung der einzelnen Bibliotheksmodule der Tycoon StP Bibliothek *StPEnv* ist in den einzelnen Tabellen in Anhang B.2 dargestellt.

3.3.3 Grenzen der Funktionalität

Bezüglich der entwickelten StP Tycoon Bibliothek *StPEnv* sind einige kritische Aspekte anzumerken, welche einen Einfluß auf die vorgenommene Modellierung der StP Umgebung auf der Ebene des Tycoon Systems haben. Daraus ergibt sich, daß die gewählte Modellierung auf der Tycoon Ebene nicht deckungsgleich zu der auf der StP Ebene ist.

Die hauptsächlichen Einschränkungen treten bei der Abbildung des StP Persistent Data Models mit Hilfe der StP API C Funktionen auf Module, Datenstrukturen und Operationen der Bibliothek *StPEnv* und der Interpretation der Anwendungssicht auf das Repository mit der anschließenden Konvertierung der Anwendungsdatentypen auf StP PDM Datentypen auf.

Abbildung des StP Persistent Data Models

Das in der Abbildung 2.6 wiedergegebene StP Persistent Data Model stellt den Zusammenhang der existierenden StP PDM Datentypen auf der Grundlage von Vererbungsbeziehungen dar, wobei jeder PDM Datentyp in einer Subtypbeziehung zu einem anderen Typen steht. Desweiteren besitzen alle PDM Datentypen eine Anzahl von Attributen: spezielle oder, im Rahmen der Vererbungsbeziehungen zwischen PDM Datentypen, gemeinsame Attribute. Leider hat es die Firma IDE versäumt, diesen Sachverhalt in der Systemdokumentation detailliert wiederzugeben, so daß an dieser Stelle keine weiteren Aussagen über die genaue Ausprägung der Vererbungsbeziehungen getätigt werden können.

Entitäten der StP PDM Datenypen werden in der relationalen Datenbank Sybase abgelegt und sind, wie in [Nietsch 96] detailliert beschrieben, unter anderem durch den Einsatz der C Funktionen des StP API zugreifbar. Sowohl auf die gesamte Struktur eines PDM Objekts als auch auf seine einzelnen Attribute kann zugegriffen werden. Allerdings lassen die API Operationen keine Rückschlüsse auf die Vererbungscharakteristika referenzierter Objekte zu, da der Repositoryzugriff mit Hilfe spezialisierter, d.h. nicht generischen, C Funktionen realisiert wird.

Da die StP API C Funktionen keinen Rückschluß zulassen, in welcher Weise das Persistent Data Model auf sie abgebildet worden ist, ist es fraglich, ob man ausgehend von den StP API Operationen im Rahmen eines *Re-Engineering* Vorgehens die tatsächlich vorliegenden Modellierungskomponenten auf der StP Ebene ausdrücken kann, um sie anschließend auf der Tycoon Ebene zu übernehmen.

Konvertierung von Anwendungs- auf StP PDM Datentypen

Wie in Abschnitt 2.3.4 erläutert, wird die Verbindung zwischen Datentypen von StP Werkzeugen und denen des Persistent Data Models mit Hilfe der Abbildungsdatei `omt.app.types` realisiert. Diese Datei enthält jedoch ausschließlich Einträge, die spezifizieren, auf welche CASE Objekttypen die Anwendungsdatentypen abzubilden sind. Die Semantik, die hinter dieser Abbildung steht, ist mit Hilfe der StP API Funktionen nicht nachzuvollziehen und folglich nicht auf die Tycoon Bibliothek *StPEnv* zu übertragen.

So können z.B. innerhalb des CASE Werkzeugs StP/OMT mit einem grafischen Editor entwickelte Komponenten von Modellen mit Hilfe der Datei `omt.app.types` auf eine Anzahl gültiger Repositoryobjekte abgebildet werden. Die Application Program Interface Operationen leisten diesen Dienst nicht, da sie es nur ermöglichen, isolierte PDM Objekte im Repository zu generieren oder auf sie zuzugreifen. Eine Modellierung von gültigen Anwendungsdatentypen kann deshalb auf der Tycoon Ebene durch die API Funktionen nur eingeschränkt erfolgen.

4. Tycoon Statecharts zur Modellierung dynamischen Systemverhaltens

Dynamisches Systemverhalten bezeichnet, wie in Abschnitt 2.2.1 dargestellt, die Gesamtheit der einnehmbaren Zustände und der möglichen Zustandsübergänge eines Systems unter Berücksichtigung zustandsübergangsauslösender Ereignisse. Bei der Erstellung eines Spezifikationsmodells des Systemverhaltens eines zu entwickelnden Anwendungssystems werden in vielen Softwareentwicklungsmethoden, wie z.B. in der Strukturierten Analyse oder der *Structured Systems Analysis and Design Method* (SSADM), endliche Automaten zur Spezifikation des Systemverhaltens verwendet [Fricke 95].

Ein endlicher Automat stellt ein mathematisches Modell eines Systems mit diskreten Ein- und Ausgaben dar. Zu jedem Zeitpunkt befindet sich das System in einem Zustand aus einer endlichen Menge von möglichen Zuständen, den Konfigurationen [Hopcroft, Ullman 90]. Zum Entwurf des Verhaltens von Systemen mit endlichen Zustandsmengen und diskreten Zustandsübergängen bietet die Theorie der endlichen Automaten ein adäquates Werkzeug zur Systemspezifikation.

Allerdings zeigt sich, daß bei großen, komplexen Systemen schnell die Grenzen der Modellierungsmächtigkeit endlicher Automaten erreicht werden, da sie in ihrem Umfang unhandlich und unübersichtlich werden. Eine Ergänzung um Zustandstabellen oder -matrizen kann zwar ihre Aussagekraft ergänzen, führt aber zu einer nicht weniger komplexen Darstellung des aus endlichen Automaten und Zustandstabellen oder -matrizen bestehenden Verhaltensmodells des Systems. Außerdem wird so der Vorteil einer ausschließlich grafischen Repräsentation des Verhaltens eines Systems mit endlicher Zustandsmenge aufgegeben.

Eine Erweiterung des Modells endlicher Automaten stellt die Spezifikationssprache der Statecharts dar [Harel 87]. Sie wird in diesem Kapitel dargestellt und im Rahmen dieser Arbeit innerhalb des Tycoon Systems exemplarisch in einer textuellen Repräsentation implementiert. Schwerpunktmäßig werden Statecharts, wie in Abschnitt 2.2.2 erläutert, zur grafischen Modellierung und Spezifikation des Verhaltens ereignisgesteuerter Systeme eingesetzt [Huizing, de Roever 91], die in einer fortwährenden Interaktion mit ihrer Systemumgebung stehen. Solche reaktiven Systeme sind z.B. Steuerungseinrichtungen für Flugzeuge oder Kraftwerke oder das in Abschnitt 2.2.2 eingeführte Beispiel der Fernsehersteuerung. Für die Modellierung rein statischer Systeme, wie z.B. eines *Repositorys*, sind Statecharts nicht geeignet.

Ausgehend von der Statechart Originalnotation aus [Harel 87] sind sowohl die Statechart Syntax als auch die Semantik in vielen Forschungsarbeiten verschieden dargestellt und erweitert worden. In [v.d. Beeck 94] ist ein Vergleich und eine Bewertung einiger unterschiedlicher Statechartvarianten enthalten.

Grundlage der in diesem Kapitel vorgestellten Tycoon Statechart Syntax innerhalb einer Tycoon Umgebung für eine textuelle statechartbasierte Modellierung dynamischen Systemverhaltens als Basis für die in Abschnitt 5.4 dargestellte Tycoon Validierungsumgebung sind die Statechartelemente des dynamischen Modells der in Abschnitt 2.2 dargestellten Softwareentwicklungsmethode OMT. Es handelt sich bei den von der OMT Methode im Rahmen des dynamischen Modells verwendeten Statechartelementen um eine Teilmenge der in [v.d. Beeck 94] verglichenen Syntaxelemente der zahlreichen Statechartvarianten.

Diese Teilmenge wird im Rahmen des exemplarischen Aufbaus der statechartbasierten Tycoon Modellierungsumgebung zur Spezifikation dynamischen Systemverhaltens auf die in der Abbildung 4.1 in der Notation des OMT Objektmodells [Rumbaugh et al. 91] dargestellten Statechart Syntaxelemente weiter eingeschränkt.

4.1 Statecharts als strukturierte endliche Automaten

Zustände und Zustandsübergänge stellen die Grundelemente endlicher Automaten dar. Zustandsübergänge, auch als Transitionen bezeichnet, arbeiten auf einer endlichen Menge von Eingabesymbolen aus einem Eingabealphabet und überführen einen Zustand in einen Nachfolgezustand [Hopcroft, Ullman 90]. Formal sind Zustandsübergänge als eine Übergangsfunktion von Quellzustand und Eingabezeichen in einen Zielzustand definiert. Überführt eine Folge von Eingabezeichen den endlichen Automaten von einem Startzustand in einen Endzustand, so wird das Eingabewort vom Automaten akzeptiert. Es gehört somit zur Sprache dieses endlichen Automaten. Neben diesen akzeptierenden endlichen Automaten gibt es endliche Automaten, deren Zustände (Moore Maschine) oder Zustandsübergänge (Mealy Maschine) mit der Ausgabe von Zeichen assoziiert sind.

Bildlich dargestellt entspricht ein endlicher Automat einem gerichteten Graphen, bei dem die Knoten den Zuständen und die gerichteten Kanten (Pfeile) zwischen den Knoten den Transitionen des endlichen Automaten entsprechen. Die Eingabesymbole werden an die Pfeile notiert. Bei einer Moore Maschine werden zusätzlich noch die Ausgabezeichen an die entsprechenden Zustände geschrieben, mit Zustandsübergängen assoziierte Ausgabezeichen werden bei der Mealy Maschine an der entsprechenden Transition vom Eingabezeichen durch einen Schrägstrich getrennt dargestellt.

Statecharts erweitern das Modell endlicher Automaten um die Konzepte der Dekomposition von Zuständen und Orthogonalität von Zuständen und Zustandsübergängen [Glinz 95]. Bei Statecharts werden Eingabezeichen als Ereignisse bezeichnet und Ausgabezeichen als Aktionen. Sowohl Ereignisse als auch Aktionen sind mit Transitionen verknüpft. Allerdings können auch, wie in Abschnitt 4.3.2 dargestellt, Aktionen mit Zuständen assoziiert werden. Ereignisse lösen Zustandsübergänge aus, zusätzlich können Bedingungen Zustandsübergänge einschränken.

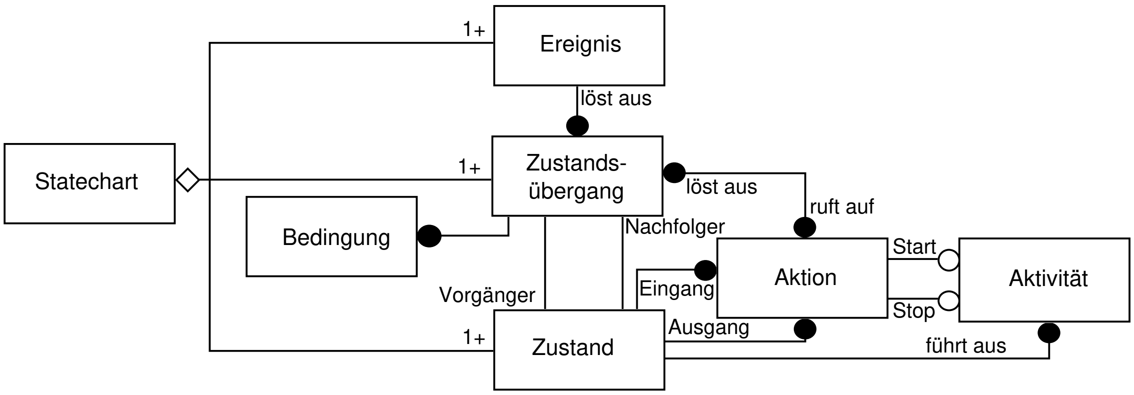


Abbildung 4.1: Statechart Metamodell

Wie in der Abbildung 4.1 wiedergegeben, werden im Rahmen der exemplarischen Implementierung der Tycoon Umgebung *StatechartEnv* zur Modellierung dynamischen Systemverhaltens in einer Tycoon Statechart Repräsentation (im folgenden auch als Tycoon Statecharts bezeichnet) die folgenden Statechart Syntaxelemente betrachtet:

- Zustand
- Zustandsübergang zwischen Zuständen
- Ereignis, sowohl externes Ereignis als auch internes Ereignis, das Zustandsübergänge auslöst
- Aktion zum Generieren interner Ereignisse als auch zum Auslösen externer Ereignisse
- Bedingung zur Einschränkung von Zustandsübergängen
- Aktivität als zeitandauernde Handlung mit Start und Stop Aktion

Eine detaillierte, informelle Darstellung dieser Statechartelemente ist in [Harel 87] enthalten. Innerhalb dieser Arbeit erfolgt eine Erläuterung der einzelnen Komponenten in den Abschnitten über ihre textuelle Tycoon Repräsentation weiter unten in diesem Kapitel.

Ein Statechart aggregiert Zustände, Zustandsübergänge und Ereignisse. Jeder Zustandsübergang besitzt einen Vorgänger- und Nachfolgezustand. Ein Ereignis kann einen oder mehrere Zustandsübergänge auslösen, falls eine für das Stattfinden eines Zustandsübergangs definierte Bedingung gültig ist. Ein Zustandsübergang kann eine oder mehrere Aktionen umfassen, die in der Form von intern generierten Ereignissen wiederum einen oder mehrere Zustandsübergänge auslösen können. Entsprechend der Moore Maschine bei endlichen Automaten kann ein Zustand mit einer oder mehreren Aktionen assoziiert werden. Eine oder mehrere in einem Zustand definierte Eingangsaktionen können einen oder mehrere Zustandsübergänge beim Übergang in diesen Zustand auslösen, entsprechendes gilt für Ausgangsaktivitäten beim Übergang in den Nachfolgezustand. Laut Synchronitätshypothese [Huizing et al. 88] treten Ereignisse und Zustandsübergänge zeitverzugslos auf. Eine Aktivität hingegen stellt eine zeitandauernde Handlung dar, die durch eine initiierende Startaktion und eine die Aktivität terminierende Aktion kontrolliert wird oder solange ausgeführt wird, wie die aktuelle Konfiguration des Systems den mit der Aktivität assoziierten Zustand umfaßt (vgl. Abschnitt 4.4.1).

David Harel definiert Statecharts in [Harel 87] mit der Formel:

statecharts = state diagrams + depth + orthogonality + broadcast-communication

Tiefe (*depth*) entsteht innerhalb von Zustandsdiagrammen durch Zusammenfassung oder Verfeinerung von Zuständen. Diese Strukturierung und Hierarchisierung von Zuständen erlaubt es, selbst komplexe und von der Größe her umfangreiche Zustandsdiagramme übersichtlich und für die Anwender und Entwickler durchschaubar darzustellen. Im Gegensatz zu endlichen Automaten können somit eine Vielzahl von Zuständen und Zustandsübergängen innerhalb eines von der Aussage her mit einem endlichen Automaten vergleichbaren Statecharts zusammengefaßt werden. Ein modelliertes System kann beliebig verfeinert auf unterschiedlichen Abstraktionsebenen in Teildiagrammen des Gesamtsystems dargestellt werden, was dazu führt, daß die Teildiagramme für sich selbst betrachtet übersichtlich bleiben. Allerdings kann durch die Verwendung unterschiedlicher Abstraktionsebenen bei der Diagrammverfeinerung in vielen Teildiagrammen der Überblick über das Gesamtsystem verloren gehen, was auch durch die Verwendung von computerbasierten, grafikorientierten Visualisierungswerkzeugen nur teilweise kompensiert werden kann.

Nebenläufigkeit, in [Harel 87] und in dieser Arbeit als Orthogonalität (*orthogonality*) bezeichnet, umfaßt innerhalb von Statecharts die Möglichkeit, unabhängige Systemkomponenten durch unabhängige Statecharts verfeinert darzustellen. Innerhalb der Verfeinerung von orthogonalen Zuständen können, wie in Abschnitt 4.2.2 dargestellt, nebenläufige Zustandsübergänge stattfinden.

Broadcast Kommunikation (*broadcast-communication*) ermöglicht eine Synchronisation von Zustandsübergängen. Hierzu werden, wie in Abschnitt 4.3 dargestellt, externe Ereignisse oder durch Aktionen generierte interne Ereignisse dem Statechart als Trigger seiner Zustandsübergänge global zur Verfügung gestellt. Das bedeutet, daß sowohl ein intern generiertes als auch ein externes Ereignis an beliebiger Stelle innerhalb des Statecharts einen Zustandsübergang auslösen kann.

In der Abbildung 4.2 ist exemplarisch das Verhalten eines Fernsehgerätes (*Television*) als Statechart auf oberster Abstraktionsebene in einer von der OMT Notation des dynamischen OMT Modells abweichenden, in der Literatur üblicherweise verwendeten Darstellung gezeigt.

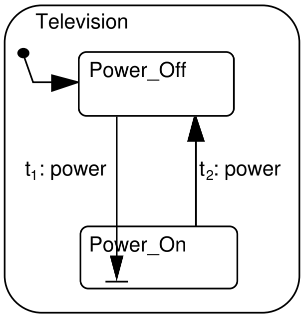


Abbildung 4.2: *Television* Statechart

In Anlehnung an das in Abschnitt 2.2.1 dargestellte OMT Metamodell spezifiziert ein Statechart innerhalb des dynamischen OMT Modells das Verhalten einer in einem OMT Objektmodell enthaltenen Klasse (hier: OMT Klasse *Television*).

Auf der in der Abbildung 4.2 wiedergegebenen Ebene der Abstraktion kann ein Objekt der Klasse *Television* die Zustände *Power_Off* und *Power_On* einnehmen, wobei der Zustand *Power_On* nach dem Konzept der Tiefe auf einer nächst tieferen Abstraktionsebene in einem weiteren Zustandsdiagramm verfeinert wird (vgl. Abbildung 4.4).

Spezifiziert ist in Abbildung 4.2 das Verhalten eines Fernsehgerätes, welches sich in zwei unterschiedlichen Betriebszuständen, ausgeschaltet (Zustand *Power_Off*) oder eingeschaltet (Zustand *Power_On*), befinden kann. Wird ein standardmäßig ausgeschaltetes Fernsehgerät eingeschaltet (Ereignis *power*), so geht die Systemkontrolle vom Zustand *Power_Off* in den Zustand *Power_On* über, wobei sich die Systemkontrolle innerhalb des verfeinerten Zustands *Power_On* in weiteren Nachfahrzuständen befindet. Eine detaillierte Darstellung der Semantik von Ausführungsschritten in Statecharts ist in Abschnitt 4.4 enthalten.

Die Konzepte der Tiefe, Orthogonalität und Broadcast Kommunikation werden in den Ausführungen über Zustände und Zustandsverfeinerungen in Abschnitt 4.2 und über die Semantik von Ausführungsschritten in Statecharts in Abschnitt 4.4 vertiefend erläutert und anhand der konkreten Implementierung der Tycoon Umgebung *StatechartEnv* für die statechartbasierte textuelle Tycoon Modellierung von dynamischem Systemverhalten dargestellt. Bei der Erläuterung der Tycoon Implementierung wird insbesondere auf die Zusicherung der syntaktischen und semantischen Integritätsbedingungen eingegangen, die teilweise durch Typisierung und teilweise durch explizit ausprogrammierte Prüfungen erreicht wird.

4.2 Zustände und Zustandsverfeinerungen

Nach dem in Abschnitt 4.1 eingeführten Konzept der Tiefe kann jeder Zustand eines Statecharts auf beliebig tief verfeinerten Abstraktionsebenen Unterzustände besitzen. Dabei sind unterschiedliche Arten der Verfeinerung von Zuständen möglich.

In der Abbildung 4.3 sind mit Hilfe der Notation des OMT Objektmodells die Generalisierungseigenschaften von Statechart Zuständen dargestellt. Generalisierung steht innerhalb der OMT Objektmodellierung für eine Relation zwischen einer Klasse und einer oder mehrerer spezialisierter Unterklassen dieser Klasse [Rumbaugh et al. 91].

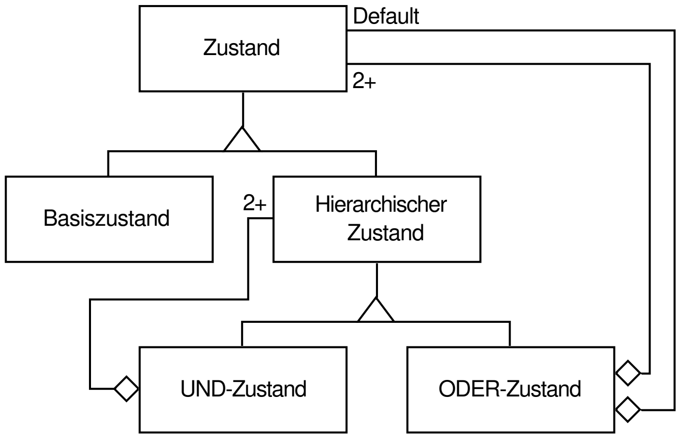


Abbildung 4.3: Generalisierungseigenschaften von Zuständen

Im Rahmen der Tycoon Repräsentation von Statecharts ist ein Zustand als Oberklasse eines Basiszustands oder eines hierarchischen Zustands anzusehen, wobei seine Unterklassen disjunkt sind. Das bedeutet, daß ein Objekt der Klasse Basiszustand nicht die Ausprägung eines Objekts der Klasse Hierarchischer Zustand annehmen kann. Ebenso kann ein Objekt der Klasse Hierarchischer Zustand nicht in der Ausprägung eines Objekts der Klasse Basiszustand auftreten. Ein hierarchischer Zustand stellt für die Klassen UND-Zustand und ODER-Zustand die Oberklasse dar, wobei im Rahmen einer Aggregationsbeziehung ein UND-Zustand mindestens zwei hierarchische Zustände umfassen muß. Ein ODER-Zustand beinhaltet mindestens zwei Objekte der Klasse Zustand, die ihrerseits wiederum nach den eben erläuterten Generalisierungseigenschaften von Zuständen, wie in der Abbildung 4.3 dargestellt, verfeinert werden können. Ein Zustand eines ODER-Zustands ist als Defaultzustand auszuzeichnen (vgl. Defaultzustand bei der Generierung einer Tycoon Repräsentation eines ODER-Zustands in Abschnitt 4.2.2).

Der in Abbildung 4.2 dargestellte *Television* Statechart auf oberster Abstraktionsebene umfaßt die folgenden Zustände:

- *Power_Off* als Basiszustand.
- *Power_On* als hierarchischer Zustand, wobei auf dieser Abstraktionsebene nicht zu erkennen ist, ob es sich um einen UND-Zustand oder einen ODER-Zustand handelt.
- *Television* als hierarchischer Zustand (ODER-Zustand) mit dem Defaultzustand *Power_Off* und dem Zustand *Power_On*.

Eine Darstellung des in der Abbildung 4.2 auf oberster Abstraktionsebene unverfeinert wiedergegebenen Zustands *Power_On* innerhalb des *Television* Statecharts auf einem niedrigeren Abstraktionsniveau ist in der Abbildung 4.4 enthalten.

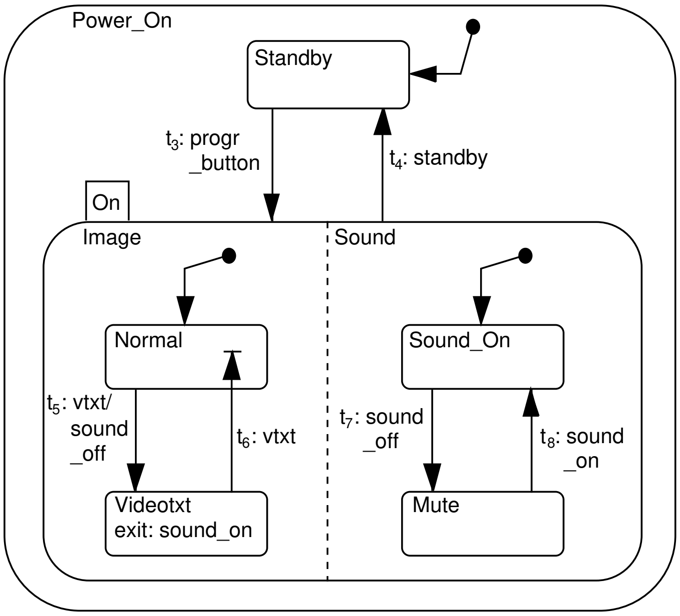


Abbildung 4.4: Verfeinerter Zustand *Power_On*

Spezifiziert wird in Abbildung 4.4 das Verhalten eines eingeschalteten Fernsehgerätes, welches sich standardmäßig in einer Betriebsbereitschaft (Zustand *Standby*) befindet. Durch das Auslösen eines externen Ereignisses mit einer Fernbedienung (OMT Klasse *Remote-Control*, vgl. Abschnitt 2.2.2), z.B. durch das Drücken einer Programmtaste (Ereignis *progr_button*), wird der Fernseher eingeschaltet und zeigt ein entsprechendes Bild (Zustand *Image* innerhalb des verfeinerten UND-Zustands *On*). Orthogonal zum Bild wird der Ton (Zustand *Sound*) des betreffenden Programms eingeschaltet. Durch das Ereignis *standby* wird das Fernsehgerät jederzeit, unabhängig davon, in welchem Nachfahrzustand des Zustands *On* sich die Systemkontrolle befindet, in die Betriebsbereitschaft zurückversetzt.

Wie eine adäquate Tycoon Repräsentation dieses grafisch modellierten Statecharts innerhalb des Tycoon Systems vorgenommen werden kann, wird in den folgenden Abschnitten exemplarisch wiedergegeben. Eine detaillierte Darstellung der Tycoon Repräsentation des in Anhang C.1 komplett grafisch spezifizierten Anwendungsbeispiels der Fernsehersteuerung ist in Anhang C.2 aufgeführt.

Innerhalb der exemplarisch betrachteten Tycoon Umgebung *StatechartEnv* zur Modellierung dynamischen Systemverhaltens in einer statechartadäquaten Tycoon Repräsentation entspricht ein Zustand dem mit dem Datentypen *Transition* wechselseitig rekursiven Datentypen *State*, dessen Definition im folgenden wiedergegeben ist:

```
Let Rec State <:Ok =
  Tuple
    label :String
    var father :optional.T(State)
    transitions :Transitions
    actionsWhenEntering :Actions
    actionsWhenLeaving :Actions
    throughoutActivities :Activities
    case basic
    case compositeAnd with children :States
    case compositeOr with
      children :States  defaultState :State
  end (* Tuple *)
and States <:Ok = set.T(State)
and Transition <:Ok =
  Tuple
    ...
  end (* Tuple *)
```

Neben seiner Bezeichnung (Attribut *label*) beinhaltet ein Tycoon Objekt vom Typ *State* eine Menge von Transitionen zu Nachfolgezuständen (Attribut *transitions*), die beim Anlegen eines Zustands als leere Menge übergeben wird und bei der späteren Verbindung des Zustands mit einem Nachfolgezustand um die entsprechende Transition erweitert wird. Mit dem Zustand assoziierte Eingangs-, Ausgangsaktionen und Aktivitäten (Attribute *actionsWhenEntering*, *actionsWhenLeaving* und *throughoutActivities*) werden bei seiner Erzeugung als Parameter übergeben (vgl. Abschnitt 4.3). Die Hierarchiebeziehungen zwischen einem Zustand und seinem Vorfahr werden durch das Attribut *father* spezifiziert, wenn der entsprechende Vorfahrzustand generiert wird.

Basiszustände oder hierarchische Zustände stellen Ausprägungen dieses Datentyps dar. Ein UND-Zustand (*compositeAnd*) umfaßt neben den oben erläuterten Attributen eines Basiszustands (*basic*) eine Menge von orthogonalen Komponenten (*children*), die ebenfalls vom Typ *State* sind. Ein ODER-Zustand beinhaltet neben seinen sich gegenseitig ausschließenden Nachfahren einen davon ausgezeichneten Defaultzustand (*defaultState*), welches derjenige

Zustand ist, an den die Systemkontrolle bei einem Zustandsübergang in diesen, den Defaultzustand umfassenden verfeinerten ODER-Zustand, übergeben wird. In der Abbildung 4.4 ist z.B. dieser ausgezeichnete Defaultzustand innerhalb des ODER-Zustands *Sound* der Basiszustand *Sound_On*.

Eine Erläuterung der abstrakten Tycoon Datentypen *Transition* und *Action* erfolgt in Abschnitt 4.3. Aktivitäten werden im Kontext dieser Arbeit nicht näher erläutert. Die Generierung der Zustände in TL erfolgt *bottom-up*, d.h. von den Basiszuständen über hierarchische Zustände zum Wurzelzustand des zu spezifizierenden Systems.

4.2.1 Basiszustand

Basiszustände bilden den Zustandsraum von Statecharts [Fricke 95] und werden nicht weiter verfeinert. In der einfachsten Form besteht ein Statechart aus mindestens zwei Basiszuständen, die wie in Abschnitt 4.2.2 beschrieben zu einem ODER-Zustand zusammengefaßt werden.

Innerhalb der Tycoon Modellierungsumgebung *StatechartEnv* wird ein Basiszustand mit der Funktion *createBasicState* in einer Tycoon Repräsentation angelegt:

```
let createBasicState(stateLabel :String  actionsWhenEntering :Actions
                     actionsWhenLeaving :Actions
                     throughoutActivities :Activities) :State =
 begin
  let state = tuple_case basic of State with stateLabel
   var optional.nil(:State) newTransitionSet() actionsWhenEntering
   actionsWhenLeaving throughoutActivities end
  set.insert(:State local.StateSet state)
  state
 end
```

Der Funktion *createBasicState* werden als Parameter die Bezeichnung des Zustands (*stateLabel*), die Mengen von Eingangs- und Ausgangsaktionen (*actionsWhenEntering*, *actionsWhenLeaving*) und die Menge der mit diesem Zustand assoziierten Aktivitäten (*throughoutActivities*) übergeben. Als Rückgabewert dieser Funktion erhält der Tycoon Anwendungsentwickler ein Objekt vom Typ *State*, welches an einen TL Bezeichner gebunden werden kann. Desweiteren wird dieser Zustand intern in eine Zustandsmenge (*local.stateSet*) eingefügt, um unter anderem auszuschließen, daß zwei Zustände mit der gleichen Bezeichnung innerhalb eines Tycoon Statecharts generiert werden. Vor dem Einfügen eines Zustands in diese statechartweite Zustandsmenge mit der Tycoon Funktion *set.insert* wird überprüft, ob ein identischer Zustand schon in dieser Menge vorhanden ist. Falls ein identischer Zustand in der Menge *local.stateSet* vorliegt, wird eine Ausnahme ausgelöst.

Ein Beispiel für eine statische Bindung eines mit der Funktion *createBasicState* generierten Basiszustands *Sound_On* (vgl. Abbildung 4.4) an den Tycoon Bezeichner *soundOn* lautet:

***let** soundOn = createBasicState("Sound_On" emptySet(:Action) emptySet(:Action) emptySet(:Activity))*

Eine komplette textuelle Tycoon Repräsentation des in Anhang C.1 grafisch spezifizierten *Television* Statecharts ist in Anhang C.2 aufgeführt. Gleiches gilt für die in den folgenden Abschnitten vorgestellten Implementierungsbeispiele, die Auszüge des Gesamtbeispiels in Anhang C darstellen.

4.2.2 Hierarchischer Zustand

Ein hierarchischer Zustand eines Statecharts ist nach dem in Abschnitt 4.1 eingeführten Konzept der Tiefe rekursiv eine Dekomposition eines Zustands in einer der folgenden Formen:

- UND-Dekomposition eines Zustands innerhalb von zwei oder mehreren orthogonalen Statecharts auf einer höheren Abstraktionsebene als UND-Zustand.
- ODER-Dekomposition eines Zustands (nachfolgend auch als Nachfahren oder Nachfahrzustände bezeichnet) innerhalb eines Statecharts auf einer höheren Abstraktionsebene als ODER-Zustand.

ODER-Zustand

In der Abbildung 4.5 ist der verfeinerte ODER-Zustand *Videotxt* dargestellt.

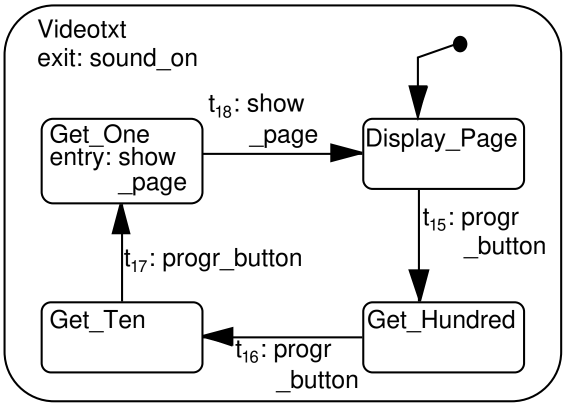


Abbildung 4.5: ODER-Zustand *Videotxt*

Er umfaßt vier sich gegenseitig ausschließende Basiszustände, welche, wie in Abschnitt 4.2.1 dargestellt, nicht weiter verfeinert sind. Der Zustand *Display_Page* ist der ausgezeichnete Defaultzustand des ODER-Zustands *Videotxt*. Verbunden werden die Zustände *Display_Page*, *Get_Hundred*, *Get_Ten*, *Get_One* und *Display_Page* durch Transitionen (vgl. Abschnitt 4.3).

Spezifiziert ist in Abbildung 4.5 das Verhalten einer Videotextsteuerung innerhalb eines Fernsehgerätes, wo per Definition beim Betreten des Videotextmodus eine Standardseite angezeigt wird (Zustand *Display_Page*) und nach dem Eingeben einer dreistelligen Seitennummer (Zustände *Get_Hundred*, *Get_Ten* und *Get_One*) durch das Drücken der entsprechenden Programmtasten auf der Fernbedienung (Ereignis *progr_button*) die gewählte Videotextseite auf dem Bildschirm dargestellt wird (Zustand *Display_Page*).

Innerhalb der Tycoon Modellierungsumgebung *StatechartEnv* wird ein ODER-Zustand mit der Funktion *createCompositeOrState* in einer Tycoon Repräsentation angelegt:

```
let createCompositeOrState(stateLabel :String  actionsWhenEntering :Actions
                           actionsWhenLeaving :Actions
                           throughoutActivities :Activities
                           children :States  defaultState :State) :State =
  if set.size(:State children) > 1
   andif set.member(:State defaultState children) then
    let state = tuple_case compositeOr of State with stateLabel
     var optional.nil(:State) newTransitionSet() actionsWhenEntering
     actionsWhenLeaving throughoutActivities children defaultState end
    iter.forEach(:State set.elements(children) fun(st :State) :Ok
     if set.member(:State st local.stateSet)
      andif optional.null(st.father) then
       st.father := optional.new(state)
     else
      raise childrenError end
     end)
    set.insert(:State local.stateSet state)
    state
  else
   raise argumentError end
  end
```

Der Funktion *createCompositeOrState* werden als Parameter die Bezeichnung des Zustands (*stateLabel*), die Mengen von Eingangs- und Ausgangsaktionen (*actionsWhenEntering*, *actionsWhenLeaving*), die Menge der mit diesem Zustand assoziierten Aktivitäten (*throughoutActivities*), die Menge der Nachfahrzustände (*children*) und ein ausgezeichneter Defaultzustand (*defaultState*) übergeben. Enthält die Menge der angegebenen Nachfahrzustände mehr als einen Zustand und kommt der angegebene Defaultzustand in der Menge der Nachfahrzustände vor, so kann der ODER-Zustand durch die TL Funktion *createCompositeOrState* generiert werden.

Desweiteren wird für jeden angegebenen Nachfahrzustand überprüft, ob er bereits in der Menge aller Zustände dieses Tycoon Statecharts existiert (***if*** *set.member(:State st local.stateSet)*) und sein Attribut *father* noch nicht gesetzt ist. Ist dort bereits ein Vorfahrzustand eingetragen, so wird ein Fehler ausgelöst (***raise*** *childrenError* ***end***), da ein Zustand im Rahmen dieser TL Implementierung stets nur innerhalb eines Vorfahren vorkommen darf. So werden z.B. überlappende Zustände [Harel 87] ausgeschlossen. Ein erfolgreich generierter ODER-Zustand wird in das Attribut *father* jedes Nachfahrzustands eingetragen, um eine feste hierarchische Ordnung zwischen Nachfahr- und Vorfahrzuständen zu gewährleisten.

Ein Beispiel für eine statische Bindung eines mit der Funktion *createCompositeOrState* generierten ODER-Zustands *Videotxt* (vgl. Abbildung 4.5) an den Tycoon Bezeichner *videotxt* lautet (vgl. Anhang C.2):

```
let displayPage = createBasicState("Display_Page" ...)
let getOne = createBasicState("Get_One" ...)

let stateSet = emptySet(:State)
set.insert(stateSet displayPage)
set.insert(stateSet getOne)

let videotxt = createCompositeOrState("Videotxt" emptySet(:Action)
  actionSet emptySet(:Activity) stateSet displayPage)
```

UND-Zustand

In der Abbildung 4.6 ist der verfeinerte UND-Zustand *Display* dargestellt, welcher die beiden orthogonalen ODER-Komponenten *Show_Picture* und *Show_ProgrNr* beinhaltet.

Beide orthogonalen ODER-Komponenten umfassen jeweils zwei Basiszustände, welche, wie in Abschnitt 4.2.1 dargestellt, nicht weiter verfeinert sind. Die Zustände *Display_Picture* und *Display_ProgrNr* sind die ausgezeichneten Defaultzustände der einzelnen ODER-Zustände *Show_Picture* und *Show_ProgrNr*. Zustandsübergänge zwischen beiden ODER-Komponenten dürfen nicht existieren.

Spezifiziert ist in Abbildung 4.6 das Verhalten der Bildanzeige innerhalb eines Fernsehgerätes, wo per Definition beim Eintreten in den Anzeigemodus (Zustand *Display*) eines Fernsehbildes sowohl das Programmbild (Zustand *Display_Picture*) als auch die Programmnummer (Zustand *Display_ProgrNr*) dargestellt wird. Die Systemkontrolle innerhalb einer ODER-Komponente verhält sich dabei unabhängig von der Systemkontrolle innerhalb des dazu orthogonalen ODER-Zustands. Das bedeutet, daß sich die Anzeige eines Programmbildes unabhängig von der Anzeige der entsprechenden Programmnummer innerhalb des Anzeigemodus verhält.

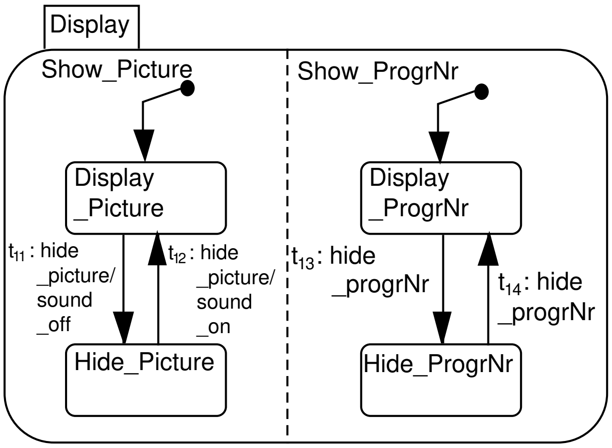


Abbildung 4.6: UND-Zustand *Display*

Innerhalb der Tycoon Modellierungsumgebung *StatechartEnv* wird ein UND-Zustand mit der Funktion *createCompositeAndState* in einer Tycoon Repräsentation angelegt, wobei ähnliche Prüfoperationen wie in der TL Funktion *createCompositeOrState* realisiert sind:

```
let createCompositeAndState(stateLabel :String  actionsWhenEntering :Actions
                            actionsWhenLeaving :Actions
                            throughoutActivities :Activities
                            children :States) :State =
  if set.size(:State children) > 1 then
    let state = tuple_case compositeAnd of State with stateLabel
      var optional.nil(:State) newTransitionSet() actionsWhenEntering
      actionsWhenLeaving throughoutActivities children end
    iter.forEach(:State set.elements(children) fun(st :State) :Ok
      if set.member(:State st local.stateSet)
        andif optional.null(st.father) then
          st.father := optional.new(state)
      else
        raise childrenError end
      end)
    set.insert(:State local.stateSet state)
    state
  else
    raise childrenError end
  end
```

Ein Beispiel für eine statische Bindung eines mit der Funktion *createCompositeAndState* generierten UND-Zustands *Display* (vgl. Abbildung 4.6) an den Tycoon Bezeichner *display* lautet (vgl. Anhang C.2):

let *displayPicture = createBasicState("Display_Picture" ...)*
let *hideProgrNr = createBasicState("Hide_ProgrNr" ...)*
...

let *showPicture = createCompositeOrState("Show_Picture" ...)*
let *showProgrNr = createCompositeOrState("Show_ProgrNr" ...)*

let *stateSet = emptySet(:State)*
set.insert(stateSet showPicture)
set.insert(stateSet showProgrNr)

let *display = createCompositeAndState("Display" emptySet(:Action)*
emptySet(:Action) emptySet(:Activity) stateSet)

Wurzelzustand

Abgeschlossen wird die Spezifikation eines Statecharts mit einem ausgezeichneten Vorfahrzustand, dem sogenannten Wurzelzustand, der über alle Verfeinerungsebenen hinweg der oberste Vorfahr aller in diesem Statechart enthaltenen Zustände ist.

Die TL Implementierung zur Generierung eines Wurzelzustandes innerhalb der *Statechart-Env* Umgebung zur Modellierung dynamischen Systemverhaltens (hier: Wurzelzustand *Television* aus der Abbildung 4.2) folgt der Implementierung der oben dargestellten Funktion *createCompositeOrState*, mit dem Unterschied, daß die Funktion *createRootState* den so erzeugten Zustand als Wurzelzustand des Statecharts intern kennzeichnet.

Ein Beispiel für eine statische Bindung eines mit der Funktion *createRootState* generierten UND-Zustands *Television* an den Tycoon Bezeichner *television* lautet (vgl. Anhang C.2):

let *powerOff = createBasicState("Power_Off" ...)*
let *powerOn = createCompositeOrState("Power_On" ...)*

let *stateSet = emptySet(:State)*
set.insert(stateSet powerOff)
set.insert(stateSet powerOn)

let *television = createRootState("Television" emptySet(:Action)*
emptySet(:Action) emptySet(:Activity) stateSet powerOff)

Die Generierung von mehr als einem Wurzelzustand innerhalb eines Tycoon Statecharts darf nicht erfolgen und wird daher mit dem Auslösen einer Ausnahme abgebrochen.

4.3 Zustandsübergänge

Ein Zustandsübergang überführt im einfachsten Fall [Huizing et al. 88] einen Zustand in einen Nachfolgezustand. Innerhalb dieser Arbeit werden Zustandsübergänge auf diesen einfachsten Fall eingeschränkt. Die semantischen Konsequenzen aus verzweigten Transitionen, die einen Quellzustand mit mehreren Nachfolgezuständen oder mehrere Quellzustände mit einem Nachfolgezustand verbinden, werden hier nicht untersucht. Ein Zustandsübergang wie z.B. die Transition t_6 zwischen dem Zustand *Videotxt* und seinem Nachfolgezustand *Display* in der verfeinerten Darstellung des Zustands *Image* in Anhang C.1.2 kann Zustände unterschiedlicher Hierarchieebenen verbinden.

Innerhalb der exemplarisch betrachteten Tycoon Umgebung *StatechartEnv* zur Modellierung dynamischen Systemverhaltens in einer statechartadäquaten Tycoon Repräsentation entspricht ein Zustandsübergang dem mit dem Datentyp *State* wechselseitig rekursiven Datentyp *Transition*, dessen Definition im folgenden wiedergegeben ist:

```
Let Rec State <:Ok =
  Tuple
    ...
  end
and States <:Ok = set.T(State)
and Transition <:Ok =
  Tuple
    source :State
    target :State
    trigger :Events
    condition :States
    actions :Actions
  end
and Transitions <:Ok = set.T(Transition)
```

Als Attribute besitzt ein Zustandsübergang vom Typ *Transition* einen Quell- und einen Nachfolgezustand (Attribute *source* und *target*), eine Menge mit dem Zustandsübergang assoziierter Ereignisse (Attribut *trigger*) und eine Menge von Aktionen (Attribut *actions*). Desweiteren kann ein Zustandsübergang durch eine Bedingung (Attribut *condition*) in der Form einer Zustandsmenge eingeschränkt werden. Diese Bedingung läßt einen Zustandsübergang nur dann stattfinden, wenn die in dieser Bedingung spezifizierten Zustände zum Zeitpunkt des möglichen Zustandsübergangs in der aktuellen Systemkonfiguration enthalten sind (vgl. Abschnitt 4.4.1).

Innerhalb der Tycoon Modellierungsumgebung *StatechartEnv* wird ein, mit Aktionen (vgl. Abschnitt 4.3.2) assoziierter Zustandsübergang, mit der Funktion *createTransitionWithActions* in einer Tycoon Repräsentation angelegt:

```
let createTransitionWithActions(source, target :State
                                trigger :Events
                                actions :Actions) :Transition =
  if not(list.empty(:Event trigger))
    andif not(set.empty(:Action actions)) then
      let transition = tuple source target trigger
        newConditionSet() actions end
      set.insert(:Transition source.transitions transition)
      transition
  else
    raise argumentError end
  end
```

Der Funktion *createTransitionWithActions* werden als Parameter der Quellzustand (*source*), der Nachfolgezustand (*target*), eine Menge von Ereignissen (*trigger*), welche die auslösenden Ereignisse des Zustandsübergangs darstellen, und eine Menge mit dem Zustandsübergang assoziierter Aktionen (*actions*) übergeben. Als Rückgabewert dieser Funktion enthält der Tycoon Anwendungsentwickler ein Objekt vom Typ *Transition*. Desweiteren wird die generierte Transition in die Zustandsübergangsmenge des Quellzustands (*source.transitions*) eingefügt. Sollte die der Funktion *createTransitionWithActions* übergebene Ereignisliste (*trigger*) oder Aktionenmenge (*actions*) keine Elemente enthalten, so wird ein Fehler ausgelöst (***raise*** *argumentError* ***end***), da eine korrekte Spezifikation des Zustandsübergangs nicht erfolgen kann.

Die Schnittstellen der innerhalb der Modellierungsumgebung *StatechartEnv* implementierten TL Operationen zum Generieren von Zustandsübergängen ohne assoziierte Aktionen (*createNormalTransition*) oder mit assoziierten Bedingungen (*createConditionalTransition*) etc. sind in Anhang B.1 dargestellt.

4.3.1 Ereignis

Ein Ereignis löst einen oder mehrere Zustandsübergänge innerhalb eines Statecharts aus. Dabei kann in der Bezeichnung der Transition ein zusammengesetzter logischer Ausdruck der Form einer Konjunktion, Disjunktion oder Negation von Ereignissen auftreten. Innerhalb dieser Arbeit werden Ereignisse auf primitive Ereignisse, also nicht zusammengesetzte logische Ausdrücke, eingeschränkt.

Ereignisse lassen sich unterscheiden in externe und interne Ereignisse. Externe Ereignisse sind solche, die in Anlehnung an das in Abschnitt 2.2.2 dargestellte OMT Ereignispfaddiagramm, zwischen OMT Klassen eines Systems ausgetauscht werden und von außen auf die das Verhalten dieser OMT Klassen spezifizierenden Statecharts wirken. Interne Ereignisse werden von einer mit einer Transition assoziierten Aktion innerhalb eines Statecharts bei einem entsprechenden Zustandsübergang generiert. Daraus resultiert, daß diese generierten internen Ereignisse bei der Berechnung des nächsten auszuführenden Zustandsübergangs berücksichtigt werden müssen. Solche durch interne Ereignisse ausgelösten Transitionen werden als *Micro Steps* bezeichnet [Pnueli, Shalev 91]. Die in einem *Micro Step* ausgeführte Transition kann ebenfalls interne Ereignisse generieren, die ihrerseits wiederum anderen Transitionen als Trigger dienen. Erst nach der Ausführung aller durch ein externes Ereignis generierten *Micro Steps* wird beim Vorliegen eines externen Folgeereignisses die Berechnung des nächsten auszuführenden Zustandsübergangs durchgeführt. Alle aus einem externen Ereignis resultierenden Zustandsübergänge (*Micro Steps*) innerhalb eines Statecharts ergeben zusammen einen *Macro Step* [Pnueli, Shalev 91].

Innerhalb der Tycoon Umgebung *StatechartEnv* entspricht ein Ereignis dem Datentyp *Event*, dessen Definition im folgenden wiedergegeben ist:

```
Let Rec Event <:Ok =
  Tuple
    case primitive with eventName :String
    ...
  end
and Events <:Ok = list.T(Event)
```

Als einziges Attribut besitzt ein primitives Ereignis (***case*** *primitive*) vom Typ *Event* einen Bezeichner (Attribut *eventName*).

Innerhalb der Tycoon Modellierungsumgebung *StatechartEnv* wird ein primitives Ereignis mit der Funktion *createPrimitiveEvent* in einer Tycoon Repräsentation angelegt:

```
let createPrimitiveEvent(trigger :String) :Event =
  tuple_case primitive of Event with trigger end
```

Der Funktion *createPrimitiveEvent* wird als Parameter der Ereignisbezeichner (*trigger*) übergeben. Als Rückgabewert dieser Funktion erhält der Tycoon Anwendungsentwickler ein Objekt vom Typ *Event*.

Ein Beispiel für statische Bindungen der mit der TL Funktion *createPrimitiveEvent* generierten Ereignisse *vtxt*, *sound_on* und *sound_off* (vgl. Verfeinerung des Zustands *Image* in Anhang C.1.2) an die Tycoon Bezeichner *triggerVtxt*, *triggerSoundOn* und *triggerSoundOff* lautet:

```
let triggerVtxt = createPrimitiveEvent("vtxt")
let triggerSoundOn = createPrimitiveEvent("sound_on")
let triggerSoundOff = createPrimitiveEvent("sound_off")
```

4.3.2 Aktion

In der Bezeichnung eines Zustandsübergangs (vgl. Transition t_5 in Anhang C.1.2) kann eine Menge von Ereignissen spezifiziert werden, die bei der Durchführung des Zustandsübergangs generiert werden sollen (hier: Ereignis *sound_off*). Desweiteren können Aktionen im Rahmen von Eingangs- oder Ausgangsaktionen mit Zuständen assoziiert werden (vgl. Ausgangsaktion *sound_on* im Zustand *Videotxt* in Abbildung 4.5 auf Seite 46). Diese intern generierten Ereignisse führen umgehend zu Zustandsübergängen in der Form der in Abschnitt 4.3.1 erläuterten *Micro Steps*.

Innerhalb der Tycoon Umgebung *StatechartEnv* entspricht eine Aktion dem Datentyp *Action*, dessen Definition im folgenden wiedergegeben ist:

```
Let Action <:Ok =
  Tuple
    actionLabel :String
    trigger :Event
    var actionPart :Fun() :Ok
 end
```

Als Attribute besitzt eine Aktion vom Typ *Action* einen Bezeichner (Attribut *actionLabel*), ein zu generierendes Ereignis (Attribut *trigger*) und einen Aktionsteil (Attribut *actionPart*), der eine parameterlose TL Funktion beinhalten kann. Detailliert wird dieser Aktionsteil in Abschnitt 5.2.2 bei der Anbindung externer Dienste an Zustandsübergänge eines Statecharts erläutert.

Innerhalb der Tycoon Modellierungsumgebung *StatechartEnv* wird eine Aktion mit der Funktion *createAction* in einer Tycoon Repräsentation angelegt:

```
let createAction(actionLabel :String  triggeredEvent :Event
                 actionPart :Fun() :Ok) :Action =
  tuple actionLabel triggeredEvent var actionPart end
```

Der Funktion *createAction* werden als Parameter ein Bezeichner der Aktion (*actionLabel*), das zu generierende Ereignis *triggeredEvent* und der Aktionsteil *actionPart* übergeben. Als Rückgabewert dieser Funktion erhält der Tycoon Anwendungsentwickler ein Objekt vom Typ *Action*.

Ein Beispiel für statische Bindungen der mit der TL Funktion *createAction* generierten Aktionen *sound_on* und *sound_off* (vgl. Verfeinerung des Zustands *Image* in Anhang C.1.2) an die Tycoon Bezeichner *actionSoundOn* und *actionSoundOff* lautet:

```
let actionSoundOn = createAction("sound_on" triggerSoundOn fun() :Ok ok)
let actionSoundOff = createAction("sound_off" triggerSoundOff fun() :Ok ok)
```

Ein komplettes Beispiel der Anbindung von Aktionen an Zustände oder Zustandsübergänge ist in Anhang C.2 enthalten.

4.4 Semantik von Ausführungsschritten

Zustandsübergänge überführen ein System von einem Zustand in einen Nachfolgezustand. Ein Ausführungsschritt überführt dabei ein System von einer konkreten Konfiguration in eine Folgekonfiguration, die den Systemzustand repräsentiert. Ausgehend von einer aktuellen Konfiguration kann stets nur eine eingeschränkte Menge von Zustandsübergängen stattfinden. In Abschnitt 4.4.2 ist die Berechnung solch einer Menge von ausführbaren Zustandsübergängen erläutert. Die in der Tycoon Umgebung *StatechartEnv* exemplarisch implementierte Statechart Semantik zur Ausführung von Statecharts auf der Basis einer gegebenen Konfiguration wird in [Pnueli, Shalev 91] detailliert vorgestellt.

4.4.1 Systemkonfiguration

Eine Systemkonfiguration beschreibt den globalen Zustand eines Statecharts, welcher sich aus einer über Abstraktionsebenen hinweg betrachteten maximal konsistenten Teilmenge aller Statechartzustände zusammensetzt. Eine Teilmenge von Zuständen eines Statecharts heißt konsistent, falls für alle Zustandspaare alternativ gilt:

- Die Zustände stehen miteinander in einer Verwandtschaftsbeziehung. Zwei Zustände sind miteinander verwandt, falls der eine Vorfahr oder Nachkomme des anderen ist, wobei zwischen beiden Zuständen mehrere Verfeinerungsebenen liegen können.
- Die Zustände sind orthogonal zueinander. Zwei Zustände sind orthogonal, falls sie nicht in einer Verwandschaftsbeziehung stehen und ihr kleinster gemeinsamer Vorfahr ein UND-Zustand ist. Der kleinste gemeinsame Vorfahr zweier Zustände ist der Zustand, bei dem sich die Hierarchiepfade von den Zuständen zur Wurzel des Statecharts hin treffen [Fricke 95].

Eine konsistente Teilmenge X von Zuständen eines Statecharts heißt maximal konsistent, falls für jeden Statechartzustand s außerhalb dieser Teilmenge von Zuständen gilt, daß die Vereinigungsmenge $X \cup \{s\}$ eine nicht konsistente Menge von Zuständen ergibt.

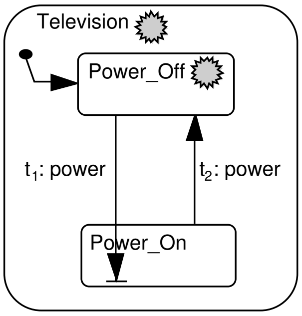


Abbildung 4.7: Ausgangskonfiguration

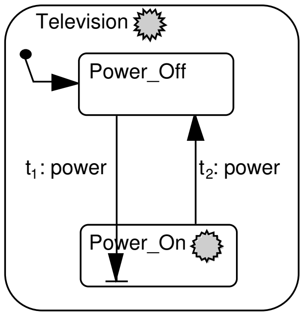


Abbildung 4.8: Folgekonfiguration

In Anlehnung an das Konzept der Markierung von Bedingungen in Netzen [Reisig 82], kann innerhalb von Statecharts ein in einer aktuellen Systemkonfiguration enthaltener Zustand grafisch ebenfalls als markiert dargestellt werden.

In der Abbildung 4.7 ist eine Ausgangskonfiguration als maximal konsistente Teilmenge von Zuständen des *Television* Statecharts dargestellt. Sie umfaßt die markierten Zustände *Television* und *Power_Off*. Ähnlich wie bei Stellen/Transitions-Netzen [Jessen, Valk 87], bei denen eine Transition nur schalten kann, wenn die davorliegenden Stellen entsprechend markiert sind und die Kapazität der dahinterliegenden Stellen durch das Schalten der Transition nicht übertroffen wird, kann in einem Statechart ein Zustandsübergang nur stattfinden, wenn der entsprechende Quellzustand in der aktuellen Systemkonfiguration enthalten ist. Ausgehend von der in der Abbildung 4.7 dargestellten Ausgangskonfiguration $C = \{Television, Power\_On\}$ wird beim Eintreten des Ereignisses *power* die in der Abbildung 4.8 gezeigte Folgekonfiguration erreicht, wobei die Konfiguration über diese Abstraktionsebene hinaus die Defaultzustände der weiter verfeinerten Zustände des Zustands *Power_On* umfaßt (vgl. Abbildung 4.4 auf Seite 43). Die in der Abbildung 4.8 dargestellte Folgekonfiguration C' beinhaltet die Zustände *Television*, *Power_On* und *Standby*.

Angelegt wird eine Ausgangskonfiguration innerhalb der Tycoon Umgebung *StatechartEnv* auf der Basis des in Anhang C.2 dargestellten Tycoon *Television* Statecharts mit der Funktion *setConfiguration*:

```
...
let stateSet = emptySet(:State)
set.insert(stateSet tv)
set.insert(stateSet powerOff)

television.setConfiguration(stateSet)
```

4.4.2 Schrittkonstruktion

Wie in Abschnitt 4.4.1 dargestellt, kann in einer aktuellen Konfiguration stets nur eine bestimmte Menge von Zustandsübergängen stattfinden. Die folgenden Bedingungen schränken die Menge der ausführbaren Transitionen ein [Day 93]:

- Ein Zustandsübergang kann stattfinden, falls sich die Systemkontrolle in dem Quellzustand des Zustandsübergangs befindet. Das bedeutet, daß dieser Quellzustand in der aktuellen Systemkonfiguration enthalten sein muß. Liegt das zustandsübergangsauslösende Ereignis vor, so kann das System in eine Folgekonfiguration übergehen.
- Innerhalb eines ODER-Zustands kann nur ein Zustandsübergang zu einem Zeitpunkt stattfinden.
- Innerhalb eines UND-Zustands können orthogonale Zustandsübergänge simultan innerhalb der einzelnen UND-Komponenten stattfinden.

Die Menge der ausführbaren Transitionen ist abhängig von den externen Ereignissen und den innerhalb eines Ausführungsschritts generierten internen Ereignissen, da jede Transition, wie in Abschnitt 4.3 dargestellt, zustandübergangsauslösende Ereignisse durch Aktionen generieren kann, die ihrerseits wiederum zu einer Folge von Zustandsübergängen (*Micro Steps*) führen können.

Ein Ausführungsschritt (*Macro Step*) ist eine maximale Folge solcher ausführbarer *Micro Steps*, an deren Ende sich das System in einem stabilen Zustand befindet, in dem keine internen Ereignisse mehr generiert werden und die nächste Systemkonfiguration erst wieder durch einen *Macro Step* auf der Basis eines externen Ereignisses erreicht werden kann. Solch eine stabile Konfiguration wird als eine dauerhafte Konfiguration bezeichnet. Während der Abfolge der *Micro Steps* werden keine externen Ereignisse als Trigger für Zustandsübergänge zugelassen, die eine Konfiguration in eine Folgekonfiguration überführen könnten. Diese Bedingung beruht auf der den Statecharts zugrungeliegenden Synchronitätshypothese, die besagt, daß ein System immer erst seine vollständige Reaktion auf ein externes Ereignis inklusive aller generierten internen Ereignisse durchführen kann, bevor das nächste externe Ereignis tatsächlich eintritt [Pnueli, Shalev 91].[1]

Zur Bestimmung der Zustandsübergänge, die beim Vorliegen externer Ereignisse und durch Aktionen generierter interner Ereignisse ausgeführt werden können, gilt für die Menge aller Transitionen T, eine die in Abschnitt 4.4.1 dargestellten Bedingungen für gültige Systemkonfigurationen erfüllende aktuelle Systemkonfiguration C und eine Menge von externen Ereignissen I eines Statecharts die Funktion *En* (*Enabling Function*) [Leveson et al. 94]:

$$\mathit{En(T,C,I)} = \mathit{relevant(C)} \cap \mathit{consistent(T)} \cap \mathit{triggered(I} \cup \mathit{generated(T))}$$

Hierbei bezeichnet nach [Harel et al. 90]

- *relevant(C)* die Menge der Transitionen, deren Quellzustände sich in der aktuellen Konfiguration C befinden;
- *consistent(T)* die Menge der Transitionen aus T, die paarweise miteinander konsistent sind;[2]
- *triggered(E)* die Menge der Zustandsübergänge, deren Trigger in der Menge der vorliegenden Ereignisse E enthalten sind;
- *generated(T)* die Menge der durch die Aktionen der Transitionen der Menge T generierten internen Ereignisse.

[1] In der Originalformulierung der Statechartsemantik [Harel 87] bestimmen sowohl die Menge der externen Ereignisse zu Beginn eines Ausführungsschritts als auch die Menge der in einem *Macro Step* generierten internen Ereignisse die Transitionen, die zu einem Zustandsübergang führen können. Das bedeutet, daß die Ereignisse für die Länge eines Ausführungsschrittes vorliegen, nicht aber im nächst folgenden Schritt berücksichtigt werden. Eine neue Menge externer Ereignisse muß vorhanden sein, um den nächsten *Macro Step* auszulösen.

[2] Eine Transition ist konsistent mit sich selbst. Desweiteren sind zwei Transitionen konsistent miteinander, wenn die Arenen beider Transitionen, also der ODER-Zustand, der sowohl den Quell- als auch den Zielzustand der Transition enthält, zueinander orthogonal sind (vgl. Abschnitt 4.4.1).

Ein konkretes Beispiel zur Berechnung von Ausführungsschritten anhand der *Enabling Function En* ist in [Pnueli, Shalev 91] enthalten. Im Rahmen der Implementierung der in Kapitel 5.4 vorgestellten Tycoon Modellierungs- und Validierungsumgebung wird innerhalb der Tycoon Umgebung *StatechartEnv* die dargestellte Semantik von Ausführungsschritten als Grundlage der Funktion *simulate* zur Ausführung von Statecharts benutzt (vgl. Anhang C.2).

5. Validierung dynamischen Systemverhaltens

Die Validierung eines spezifizierten dynamischen Modells dient zur Überprüfung, ob das modellierte Systemverhalten den Anforderungen und Erwartungen des Anwenders entspricht. Eine Methode zur Analyse von Systemen und hierbei speziell zur Validierung des modellierten Systemverhaltens ist die Simulation auf der Basis charakteristischer Anwendungsszenarios [Guttenberg 91]. Die szenariobasierten Validierungsexperimente werden am Modell des dynamischen Systemverhaltens durchgeführt, um Rückschlüsse auf das zukünftige Verhalten des zu realisierenden Anwendungssystems ziehen zu können. Bei einer korrekten Abbildung des antizipierten Verhaltens des zu spezifizierenden Realsystems auf das zu validierende Modell lassen sich die Abläufe des realen, dynamischen Systems im Modell nachvollziehen und Kenntnisse über das Modellverhalten sammeln, die in Grenzen Rückschlüsse auf das Verhalten des zu entwickelnden Anwendungssystems in seiner realen Systemumgebung erlauben [Krüger 74].

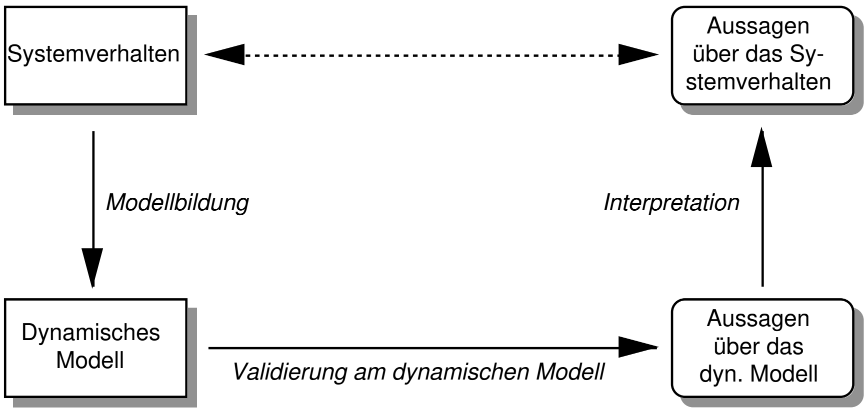


Abbildung 5.1: Systemanalyse durch Validierung

Das Vorgehen bei der Validierung von Systemen ist in der Abbildung 5.1 dargestellt. Für das Verhalten des zu entwickelnden Systems wird, wie in Abschnitt 2.2.2 vorgestellt, ein dynamisches Modell aller relevanten Zustände und Zustandsübergänge erstellt. Daraufhin wird die Validierung an diesem dynamischen Modell durchgeführt, um zu Validierungsergebnissen zu gelangen, mit deren Hilfe Aussagen über das Verhalten des Modells getroffen werden können. Eine anschließende Interpretation dieser Aussagen über das dynamische Modell ermöglicht Schlußfolgerungen über das reale Systemverhalten. Diese können dann, wie in Abschnitt 5.1 dargestellt, in den Softwareentwicklungsprozeß einfließen, um anhand der Validierungsergebnisse eine mit dem Anwender zusammen entwickelte optimale Gestaltung des dynamischen Modells zu erreichen.

Validierungsmodelle können in unterschiedliche Kategorien eingeteilt werden [Page 91]:

- Erklärungsmodelle, welche die Struktur des zu realisierenden Anwendungssystems so detailliert wiedergeben, daß sie zur Erklärung des antizipierten Systemverhaltens herangezogen werden können.
- Prognosemodelle, um zukünftige Systemzustände des zu realisierenden Anwendungssystems unter gegebenen Annahmen abzuschätzen.
- Gestaltungsmodelle als Entscheidungshilfen beim Entwurf von Systemen.
- Optimierungsmodelle zur Ermittlung und Realisierung optimaler Systemzustände und Zustandsübergänge.

Innerhalb dieser Arbeit wird die Validierung eines dynamischen Modells sowohl als eine Prognose-, Gestaltungs- und Optimierungsmethode betrachtet, um das endgültige System den Anforderungen der Anwender entsprechend realisieren zu können, als auch als Erklärungsmöglichkeit zwischen Entwicklern und Anwendern innerhalb eines Softwareentwicklungsprojekts, um für beide Rollen eine Durchschaubarkeit des zu entwickelnden Systems zu erreichen.

Eine in diesem Kapitel vorgestellte Validierungstechnik ist die in Abschnitt 5.3.2 dargestellte szenariobasierte Simulation eines dynamischen Modells, das entweder in einer textuellen Repräsentation vorliegt (vgl. *Television* Statechart in Anhang C.2) oder welches als Validierungsprototyp aus seiner StP/OMT Repräsentation generiert wird. Beide Wege führen zu einer Tycoon Repräsentation des dynamischen Modells, wie sie in Kapitel 4 beschrieben ist. Dabei können an die Tycoon Repräsentation des dynamischen Modells eines Systems, wie in Abschnitt 5.2.2 erläutert, externe Dienste angebunden werden, welche als Ereignisauslöser und -empfänger eine interaktive Simulation ermöglichen.

5.1 Einordnung in den Softwareentwicklungsprozeß

Die in Abschnitt 2.2 dargestellte *Software Engineering* Methode OMT deckt alle Phasen des Softwarelebenszyklus ab [Rumbaugh et al. 91]. Im Rahmen der Validierung dynamischen

Systemverhaltens liegt der in dieser Arbeit untersuchte Bereich innerhalb der Methode OMT ausschließlich in ihrer Analysephase, da als Ziel der Validierung des dynamischen Modells am Ende der Analysephase ein valides dynamisches Modell existieren soll, welches als stabile Grundlage für die nachfolgenden Phasen eines Softwareentwicklungsprojekts gilt.

Ziel der Analyse ist es, die zu erstellende Anwendung und die Domäne, in der sie zum Einsatz kommen soll, zu verstehen und zu modellieren.

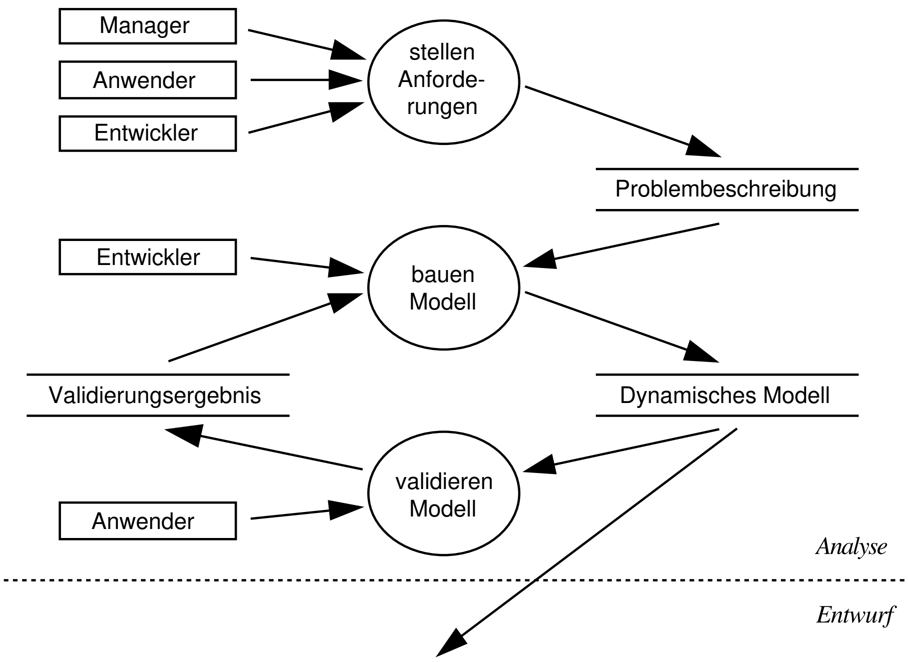


Abbildung 5.2: Einordnung der Validierung in den Modellierungsprozeß

In der Abbildung 5.2 ist der Prozeß der Validierung des dynamischen OMT Modells in die Analysephase der OMT Methode in der Notation des funktionalen OMT Modells dargestellt. Ausgehend von einer Problembeschreibung, die in einer informellen Art vorliegt und die Anforderungen der Anwender und Entwickler umfaßt, aber auch Vorgaben von Entscheidungsträgern beinhaltet, wird neben dem hier nicht als relevant betrachteten Objektmodell und dem funktionalen Modell ein dynamisches Modell erstellt. Dieses Teilmodell des gesamten Analysemodells stellt eine präzise und formalisierte Repräsentation des in der Problemstellung formulierten Systemverhaltens dar, auf dessen Basis in der anschließenden Entwurfsphase eine globale Problemlösungsstrategie für die Implementierung des Systemverhaltens entwickelt wird [Rumbaugh et al. 91]. Neben dem in Abschnitt 2.2 erläuterten OMT Objektmodell und dem funktionalen OMT Modell wird in dieser Arbeit hauptsächlich das dynamische OMT Modell untersucht.

Eine Entwicklung großer Modelle geschieht auf iterativem Wege. Zunächst wird nur eine

Untermenge des kompletten Modells erstellt, die dann solange erweitert und komplettiert wird, bis alle operationalen Aspekte der Problemstellung innerhalb des dynamischen Modells ihre Entsprechung finden. In diesem Zusammenhang wird die Validierung mit Hilfe des Anwenders in den Prozeß der Analyse eingeführt, wobei das Validierungsergebnis im nächsten Iterationsschritt beim Erstellen der nächsten Version des dynamischen Modells vom Entwickler berücksichtigt wird. Die so erstellte Folgeversion des dynamischen Modells wird mit Hilfe des Anwenders wiederum einer Validierung unterzogen.

Dieser Validierungszyklus innerhalb der Analysephase zwischen dem Anwender und dem Entwickler ist in der Abbildung 5.3 dargestellt.

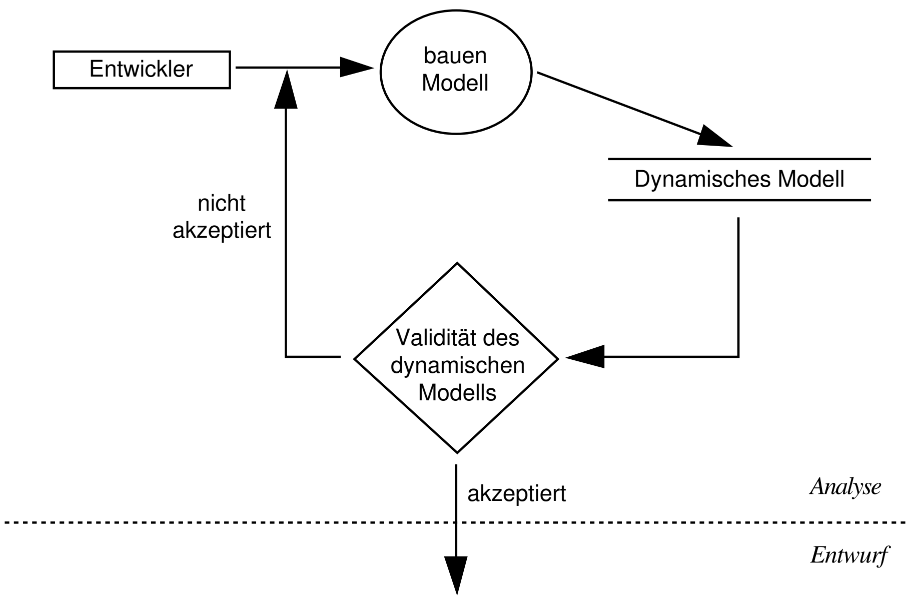


Abbildung 5.3: Validierungszyklus

Hierbei wird ein entwickeltes dynamisches Modell auf seine Validität untersucht. Validität bezeichnet die Zulässigkeit der Modellstruktur und -logik vor den Anforderungen des Anwenders [Henize 84]. Im Rahmen der Validierung des Systemverhaltens wird speziell die operationale Validität eines Modells überprüft, die im Gegensatz zur Datenvalidität ausschließlich das Modellverhalten betrachtet. Entspricht die Validität des dynamischen Modells den Erwartungen des Anwenders, so wird das erstellte und nach einigen Iterationsschritten endgültig validierte dynamische Modell als Grundlage für das weitere Vorgehen in der Entwurfsphase des Softwarelebenszyklus genommen.

Modellvalidierung kann also als ein Prozeß betrachtet werden, der ergänzend zum eigentlichen Modellierungsprozeß innerhalb der OMT Analysephase eines Anwendungsentwicklungsprojekts abläuft, um für den Entwurf ein ausreichend validiertes Modell zur Verfügung zu stellen.

5.2 Validierungstechniken

Dem Systemanalytiker und Anwender stehen mehrere Techniken zur Verfügung, um ausgehend von einem dynamischen Modell Gültigkeitsnachweise für die Erwartungskonformität des zu entwerfenden Systems durchzuführen. Diese Gültigkeitsnachweise im Rahmen der Validierung befassen sich im Gegensatz zur Verifikation von formal beweisbaren Modelleigenschaften (*"Are we building the product right?"* [Boehm 79]) mit den folgenden Problemstellungen:

- *"Are we building the right product?"* [Boehm 79]
- Ist die Spezifikation des dynamischen Systemverhaltens anwendergerecht?
- Kann eine Erwartungshaltung des Anwenders dem Systemverhalten gegenüber mit Hilfe der Systemspezifikation befriedigt werden?

Da das Untersuchungsgebiet der Validierung nicht in mathematisch beweisbarer Form vorliegt [Neufeld 93], kann eine Untersuchung hinsichtlich der Erwartungskonformität des Anwenders nur dadurch erreicht werden, daß der Anwender, wie in Abschnitt 5.1 dargestellt, in den Validierungsprozeß einbezogen wird.

Ausgehend vom CASE Werkzeug StP/OMT muß festgestellt werden, daß dieses keinerlei Validierungsmethodik für eine mit den StP/OMT Modelleditor für das dynamische OMT Modell entwickelte Spezifikation des Verhaltens eines Systems zur Verfügung stellt. Für die Überprüfung der operationalen Validität eines so entwickelten dynamischen Modells werden die folgenden Techniken unter Verwendung des Tycoon Systems als Basistechnologie zur Implementierung vorgestellt:

- Generierung eines Tycoon Validierungsprototypen aus einem vorliegenden StP/OMT dynamischen Modell.
- Interaktive Simulation auf diesem generierten Tycoon Validierungsprototypen durch den Anwender. Hierzu ist es möglich, externe Dienste an diese Tycoon Modellrepräsentation anzubinden, welche in einer ereignisauslösenden oder ereignisaufnehmenden Beziehung zum generierten Tycoon Validierungsprototypen stehen.
- Analyse von Ereignispfaden ausgehend von einem OMT Ereignispfaddiagramm (vgl. Abschnitt 2.2.2).

Sowohl die interaktive Simulation inklusive der Anbindung externer Dienste als auch die Analyse von Ereignispfaden können ebenfalls unabhängig von der Generierung einer Tycoon Repräsentation aus dem dynamischen StP/OMT Modell anhand eines innerhalb der Tycoon Modellierungsumgebung *StatechartEnv* separat entworfenen dynamischen Modells erfolgen. Dafür stellt die in Anhang B.1 wiedergegebene Modellierungsumgebung *StatechartEnv* die entsprechenden Operationen zur Verfügung.

5.2.1 Generierung validierbarer Prototypen

Ein Ziel dieser Arbeit ist die automatische Generierung eines Validierungsprototypen aus einem vorliegenden dynamischen StP/OMT Modell. Dieses modellbasierte *Prototyping* [Glinz 93], also die Generierung eines Prototypen aus der in der Form eines dynamischen Modells vorliegenden Spezifikation des Systemverhaltens, hat gegenüber der expliziten Programmierung eines Prototypen die folgenden Vorteile:

- Ein Modell beschreibt die Strukturen und Zusammenhänge in einem System. Ein programmierter Prototyp kann hingegen nur indirekt über die Analyse seines Verhaltens Aussagen über den Systemaufbau bieten.
- Modelle sind weniger aufwendig änderbar als Programmkode.
- Nach Abschluß des modellbasierten *Prototyping* unter Durchlaufen des in Abschnitt 5.1 erläuterten Validierungszyklus zwischen Anwender und Entwickler steht mit dem Modell eine komplette und abstrakte Vorgabe für die Realisierung des Anwendungssystems in den nachfolgenden Entwicklungsphasen zur Verfügung.
- Ein durch modellbasiertes *Prototyping* erzeugter Prototyp realisiert ausschließlich die im Modell berücksichtigten Anforderungen.

In der Abbildung 5.4 ist der Pfad von einem StP/OMT dynamischen Modell zu einem validierbaren Tycoon Prototypen im Rahmen des modellbasierten *Prototyping* dargestellt.

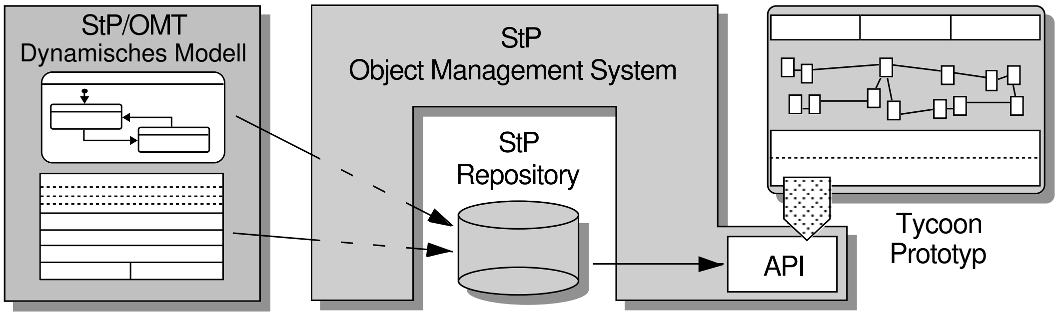


Abbildung 5.4: Modellbasiertes Prototyping

Ausgangspunkt ist hierbei eine mit dem StP/OMT Diagrammeditor für das dynamische OMT Modell entwickelte Verhaltensspezifikation eines zu entwerfenden Systems. Wie in Abschnitt 2.2.2 vorgestellt, können Zustände dieses dynamischen Modells mit Hilfe des StP/OMT Tabelleneditors z.B. um Eingangs- und Ausgangsaktionen ergänzt werden. Diese OMT Objekte eines dynamischen StP/OMT Modells, wie zum Beispiel Objekte vom StP/OMT Anwendungsdatentyp `OMTState` (Zustände), `OMTConcurrentState` (ODER-Zustände), `OMTTransition` (Zustandsübergänge), `OMTAction` (Aktionen), `OMTActivity` (Aktivitäten) etc. werden mit Hilfe der in Abschnitt 2.3.4 erläuterten StP Abbildungsdatei `omt.app.types` auf die vom StP Repository verwendeten StP PDM Datentypen in einer relationalen Struktur abgebildet. Eine Tabelle, welche die Beziehungen zwischen Objekten eines dynamischen OMT Modells und ihren entsprechenden StP/OMT Anwendungsdatentypen und deren Abbildung auf die mit ihnen assoziierten StP PDM Datentypen zeigt, ist in [IDEOMTModels 93] aufgeführt.

Beispielsweise entspricht ein Zustand (vgl. Abschnitt 4.2) innerhalb eines mit dem CASE Werkzeug StP/OMT erstellten dynamischen OMT Modells dem StP/OMT Anwendungsdatentyp `OMTState`, welcher bei der Speicherung im Repository auf ein Tupel vom StP PDM Datentyp `node` abgebildet wird.

⋮	⋮	⋮	⋮	⋮
Node	Type	Name	ScopeId	...
⋮	⋮	⋮	⋮	⋮
102	OMTState	Television	0	...
106	OMTState	Power_Off	102	...
108	OMTState	Power_On	102	...
⋮	⋮	⋮	⋮	⋮

Tabelle 5.1: Abbildung von Zuständen auf das StP Repository

In der Tabelle 5.1 ist ein Ausschnitt der relationalen StP Repositorystruktur für Tupel des StP PDM Datentypen `node` exemplarisch dargestellt. Die Verfeinerung eines Zustands erfolgt auf dieser relationalen Abstraktionsebene mit Hilfe des Attributs `ScopeId`, welches angibt, zu welchem direkten Vorfahrzustand die einzelnen `node` Tupel zugeordnet werden. Ein Wurzelzustand auf höchster Abstraktionsebene (hier: Zustand *Television*) besitzt als `ScopeId` den Wert `0`. In der Tabelle 5.1 ist der in der Abbildung 4.2 verfeinert dargestellte Zustand *Television* als Tupel des StP PDM Datentyps `node` wiedergegeben, welcher als Nachfahrzustände die Tupel `Power_On` und `Power_Off` umfaßt.

Die Ablage der StP/OMT Anwendungsdatentypen des dynamischen StP/OMT Modells im StP Repository wird, wie in Abschnitt 4.2.1 gezeigt, mit Hilfe des StP Object Management Systems realisiert. Hierbei werden die Diagrammelemente und die Elemente der zugehörigen Zustandstabellen vor der Speicherung auf das Erfüllen von Konsistenzkriterien und Integritätsbedingungen geprüft.

Innerhalb der Tycoon Modellierungs- und Validierungsumgebung dynamischen Systemverhaltens kann ein Programmierer eine Tycoon Repräsentation eines dynamischen StP/OMT Modells zwecks nachfolgender Validierung mit der TL Funktion *generateStatechart* automatisch generieren lassen:

```
import stp2Tycoon

let projdir = "/local/dbis11/software/StP/project"
let system = "nietsch"
let rootState = "Television"

let tv = stp2Tycoon.generateStatechart(projdir system rootState)
```

Für die Generierung eines Tycoon Validierungsprototypen *TV*, gebunden an den TL Bezeichner *tv*, aus den innerhalb des StP Repositorys abgelegten Objekten eines dynamischen StP/OMT Modells müssen im Rahmen der TL Funktion *generateStatechart* die folgenden Schritte durchgeführt werden:

1. Die StP Repositoryobjekte des zu validierenden dynamischen StP/OMT Modells müssen aus der Menge aller im StP Repository abgelegten Modellierungsobjekte sämtlicher StP/OMT Modelle unter Verwendung der Operationen des in Abschnitt 3.3 dargestellten StP *Gateways* anhand ihres Wurzelzustands (*Root State*) transitiv identifiziert werden. Dazu gehören Repositoryobjekte vom StP PDM Datentyp `node`, `link` und `item`, welche die oben erläuterten Objekte eines dynamischen StP/OMT Modells repräsentieren.

   ```
   let rootNode = getOnly(omsCollection.create(... omsQuery.new(
     "node[name = \'" <> rootState <> "\'" <> " && scope_node_id = 0]")))
   ...
   let stateCollection = buildScopedCollection(... rootNode ...)
   ...
   let transitionCollection = omsCollection.create(... omsQuery.new(
     "item[value != NULL && type = 501701]"))
   ...
   ```

   Mit Hilfe der in Abschnitt 2.3.5 dargestellten StP OMS Anfragen gelingt es, eine Kollektion (*stateCollection*) aller transitiv erreichbarer Zustände eines Wurzelzustands eines dynamischen StP/OMT Modells zu erstellen. Ebenso wird eine Kollektion aller Zustandsübergänge (*transitionCollection*) zwischen diesen Zuständen angelegt (vgl. StP PDM Datentyp Identifikator `501701` in Abbildung 2.9 auf Seite 24).

2. Die identifizierten StP Repositoryobjekte eines dynamischen StP/OMT Modells müssen mit Hilfe der in Kapitel 4 dargestellten TL Funktionen zu einer Tycoon Statechart Repräsentation vom Tycoon Datentyp *statechart.Chart* (vgl. Tycoon Schnittstelle *Statechart* in Anhang B.1.1) zusammengefügt werden, welche dem Anwender zur Validierung des dynamischen Systemverhaltens ausgehend von dieser Tycoon Repräsentation eines dynamischen StP/OMT Modells die in Abschnitt 5.3 vorgestellten Validierungsoperationen zur Verfügung stellt (modellbasierter Validierungsprototyp).

```
let allStates(...) = buildAllStates(stateCollection)
...
iter.forEach(... allStates fun(state ...) :Ok
  if basicState(state) then
    ...
    statechart.createBasicState(stateLabel inActions ...)
  end)
...
```

Für alle Zustände innerhalb der Kollektion *stateCollection* wird eine Tycoon Statechart Repräsentation eines dynamischen StP/OMT Modells unter Verwendung der innerhalb der Tycoon Modellierungsumgebung *StatechartEnv* exportierten Funktionen aufgebaut. Beispielsweise werden Basiszustände innerhalb der Kollektion *stateCollection* mit Hilfe der in Abschnitt 4.2.1 erläuterten TL Operation *createBasicState* generiert.

Auf diese Art werden ebenfalls Zustandsübergänge, Aktionen und Aktivitäten eines dynamischen StP/OMT Modells in eine Tycoon Statechart Repräsentation überführt.

Solch ein automatisch generierter Tycoon Validierungsprototyp dient als Grundlage der in Abschnitt 5.3 dargestellten Vorgehensweise zur modellbasierten Verhaltensanalyse von spezifizierten Systemen im Rahmen der Analysephase der OMT Methode.

Über den in Abbildung 5.4 dargestellten Pfad von einem StP/OMT dynamischen Modell zu einem validierbaren Tycoon Prototypen im Rahmen des modellbasierten *Prototyping* können mit Hilfe des StP/OMT Tabelleneditors Tycoon Kodefragmente mit Aktionen von Zuständen eines dynamischen Modells assoziiert werden. Zu diesem Zweck werden die mit dem StP/OMT Tabelleneditor in der Form von Zeichenketten spezifizierten und anschließend im StP Repository abgelegten TL Anweisungen bei der Prototyp Generierung auf der Tycoon Ebene aus dem Repository gelesen (vgl. Operationen des StP Object Management Systems für den StP Repositoryzugriff in Abschnitt 3.2). Der Tycoon *Compiler* übersetzt zur Generierungszeit des Prototypen die aus dem Repository gelesenen Zeichenketten in Tycoon Funktionen. Anschließend erfolgt zur Laufzeit eine reflektive Einbindung [Atkinson et al. 91] der vom Tycoon *Compiler* generierten Tycoon Funktionen in den Validierungsprototypen an den im StP/OMT Tabelleneditor spezifizierten Eingangs- und Ausgangsaktionen der entsprechenden Zustände des dynamischen Modells.

Dieser Vorgang der Reflektion ist ausschließlich als Erweiterungsmöglichkeit des modellbasierten *Prototyping* vorgesehen und kein Untersuchungspunkt dieser Arbeit.

5.2.2 Anbindung externer Dienste

Bei reaktiven Systemen handelt es sich stets um offene Systeme, welche in einer bidirektionalen Interaktionsbeziehung zu ihrer Systemumgebung oder einem anderen externen System stehen [Page 91]. Daher wird es im Rahmen der Validierung des dynamischen Systemverhaltens erforderlich, die Verflechtung des Systems mit seiner Systemumgebung zu berücksichtigen. So kann z.B. für Ereignisse, die in der Systemumgebung auftreten und zu einer Reaktion innerhalb des zu entwerfenden Systems führen, das daraus resultierende Systemverhalten nachvollzogen werden. Umgekehrt ist es möglich, daß innerhalb des zu validierenden Systems Ereignisse durch Aktionen generiert werden, die zu Reaktionen innerhalb der Systemumgebung oder einem anderen externen System führen.

Ein Beispiel für die Verflechtung eines zu validierenden Systems mit seiner Systemumgebung stellt die Anbindung der grafischen Benutzeroberfläche StarView an einen Validierungsprototypen (hier: Tycoon Prototyp *TV*) innerhalb der Modellierungs- und Validierungsumgebung *StatechartEnv* dar. Der externe Dienst StarView dient hierbei zur Auslösung von Ereignissen innerhalb des Tycoon Validierungsprototypen und zur Aufnahme innerhalb des Prototypen generierter Aktionen an externe Systeme.

Bei StarView handelt es sich um eine portable C++ Klassenbibliothek zur Entwicklung und Gestaltung grafischer Benutzeroberflächen [Hachmann 96]. Mit Hilfe eines Tycoon StarView *Gateways* [Geisler 95] ist es möglich, StarView GUI Elemente über TL Funktionen anzusprechen und in Tycoon Applikationen einzubinden.

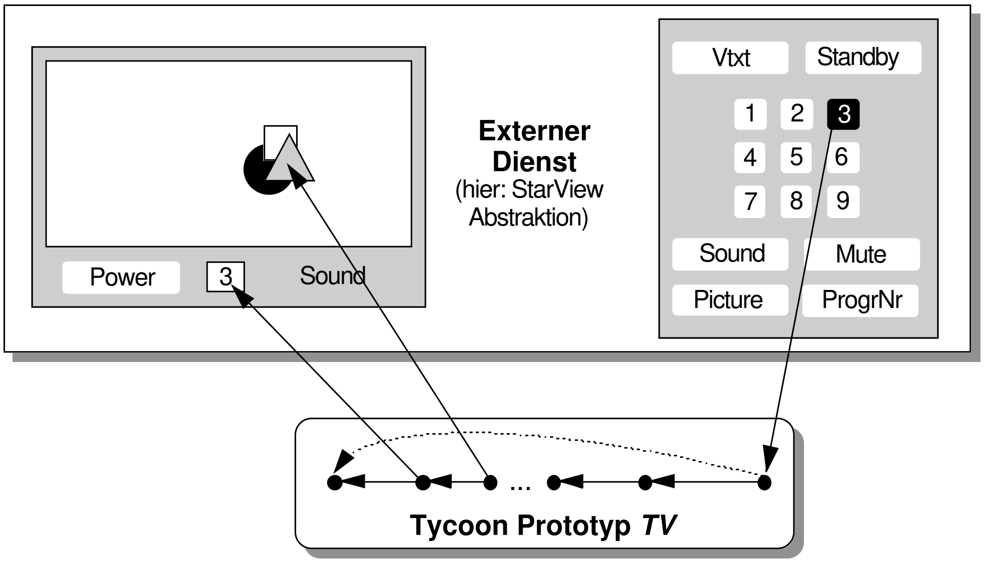


Abbildung 5.5: Anbindung externer Dienste

In der Abbildung 5.5 sind die Interaktionsbeziehungen zwischen einem zu validierenden

System (hier: der generierte Tycoon Prototyp *TV*; vgl. Abschnitt 5.2.1) und seiner Systemumgebung über die ereignisauslösende Schnittstelle einer Fernbedienung und die ereignisaufnehmende Oberfläche des Bildschirms in einer Abstraktion der entsprechenden StarView GUI Elemente dargestellt.

Das in der Abbildung 5.5 abstrahiert dargestellte Auslösen eines globalen Ereignisses erfolgt interaktiv durch den Anwender mit einem Mausklick auf eine Taste der Fernbedienung, z.B. die Programmtaste 3. Dieses globale Ereignis löst innerhalb des Tycoon Validierungsprototypen *TV* einen, wie in Abschnitt 4.4 erläutert, *Macro Step* aus, welcher durch die Generierung interner Ereignisse mehrere *Micro Steps* beinhalten kann und den Prototypen von einer Ausgangskonfiguration in eine stabile Folgekonfiguration überführt. In diesem Beispiel löst das Drücken der Programmtaste 3 innerhalb des Tycoon Prototypen *TV* durch Aktionen generierte Ereignisse an den externen StarView Dienst aus, welche die StarView Funktionen zum Anzeigen der Programmnummer und des entsprechenden Bildes aufrufen.

Realisiert wird die Verknüpfung des Tycoon Validierungsprototypen mit dem externen, ereignisauslösenden Dienst StarView mit Hilfe der TL Funktion *createRemoteControl*, welche die in Abbildung 5.5 dargestellte Fernbedienung in einem eigenen Bildschirmfenster erzeugt:

```
let createRemoteControl() :Ok =
  begin
    let cntrlWin = workWindow.new2(...)

    let progrButton3 = pushButton.new1(cntrlWin)
    ...
    let clickHandler() :Ok =
      begin
        ...
        tv.simulate(progrButton)
      end
    button.changeClickHdl(progrButton3 link.new(clickHandler))
    ...
    window.show(cntrlWin)
    window.show(progrButton3)
    ...
    application.execute()
  end
```

In diesem Auszug der Funktion *createRemoteControl* wird die Anbindung des Tycoon Validierungsprototypen *TV* über die Ereignisbehandlung (*clickHandler*) der Programmtaste 3 (*progrButton3*) realisiert. Angebunden wird der Tycoon Prototyp an die Funktion *clickHandler* mit der Funktion *tv.simulate* (vgl. Anhang B.1.1), die einen *Macro Step* innerhalb des an den Tycoon Bezeichner *tv* gebundenen *TV* Statecharts durch die Übergabe eines globalen Ereignisses (*progrButton*) auslöst.

Durch den Aufruf der StarView Ereigniskontrolle (Funktion *application.execute*) innerhalb der TL Funktion *createRemoteControl* werden die per Mausklick ausgelösten Ereignisse in einem von der Ausführung des Validierungsprototypen unabhängigen Prozeß überwacht. Bei einem Mausklick auf die Programmtaste 3 wird die Funktion *tv.simulate(progrButton)* zur Übergabe des Ereignisses *progrButton* an den Validierungsprototypen *TV* aufgerufen (vgl. Anhang C). Die Kontrolle der StarView Ereignisse existiert als nebenläufiger Prozeß zur Kontrolle des Validierungsprototypen. Auf die gleiche Art könnten mehrere externe Dienste parallel zur Auslösung externer Ereignisse an einen Tycoon Prototypen angebunden werden.

Umgekehrt können beliebige TL Funktionsaufrufe an Zustände in der Form von Eingangs- oder Ausgangsaktionen oder an Zustandsübergänge in der Form von Aktionen innerhalb einer Tycoon Statechart Repräsentation angebunden werden. Zu diesem Zweck werden an Zustände oder Zustandsübergänge im Rahmen der Modellierung des dynamischen Systemverhaltens statisch TL Funktionen als *actionPart* gebunden (vgl. Abschnitt 4.3.2):

```
...
let triggerDisplayProgrNr = tv.createPrimitiveEvent("display_progrNr")
let triggerHideProgrNr = tv.createPrimitiveEvent("hide_progrNr")
...
let actionDisplayProgrNr = tv.createAction("display_progrNr"
  triggerDisplayProgrNr  fun() :Ok window.show(showProgrNr))
...
let t14 = tv.createTransitionWithActions(hideProgrNr displayProgrNr
  triggerHideProgrNr actionDisplayProgrNr)
```

Innerhalb des Tycoon Validierungsprototypen *TV* wird als Aktionsteil die TL Funktion zum Anzeigen der Programmnummer *window.show(showProgrNr)* innerhalb des dafür vorgesehenen StarView Fensters an die Aktion *actionDisplayProgrNr* gebunden, welche mit der Transition t_{14} (entspricht dem Tycoon Bezeichner t14) zwischen den Zuständen *hideProgrNr* und *displayProgrNr* assoziiert ist.

Ausgelöst wird die Transition *t14* innerhalb des Tycoon Validierungsprototypen *TV* durch das Ereignis *triggerHideProgrNr*, wodurch die Aktion *actionDisplayProgrNr* aufgerufen wird, die das interne Ereignis *triggerDisplayProgrNr* generiert und die externe Funktion *window.show(showProgrNr)* aufruft. Die Transition t_{14} führt somit zu einem Zustandsübergang innerhalb des *TV* Prototypen, zu einer Generierung eines internen Ereignisses (zum Auslösen eines *Micro Steps*) und zum Aufruf einer Tycoon Funktion eines an den Prototyp angebundenen, externen Dienstes.

5.3 Vorgehensweisen zur modellbasierten Verhaltensvalidierung

Grundlage der in diesem Abschnitt betrachteten modellbasierten Verhaltensvalidierung ist ein in einer Tycoon Repräsentation vorliegendes dynamisches Modell (vgl. *Television* Tycoon

Statechart in Anhang C.2). In diesem Fall ist der Tycoon Statechart explizit programmiert. Wie in Abschnitt 5.2.1 vorgestellt, kann aber auch ein aus einem StP/OMT dynamischen Modell generierter Tycoon Validierungsprototyp Grundlage der modellbasierten Verhaltensvalidierung sein.

5.3.1 Statische Validierung

Im Rahmen der statischen Validierung wird untersucht, ob vom Anwender vorgegebene Systemkonfigurationen (vgl. Abschnitt 4.4.1) vom spezifizierten System eingenommen werden können. Dabei gibt der Anwender aus seiner Sicht relevante Konfigurationen vor, die das zu erstellende System in der Realität einnehmen können soll.

Stellt sich eine Systemkonfiguration als nicht einnehmbar heraus, muß der Anwender prüfen, ob die von ihm vorgegebene Konfiguration in der Realität tatsächlich eintreten kann. Ist dieses der Fall, so muß die Verhaltensspezifikation des Systems im Rahmen des in Abschnitt 5.1 vorgestellten Validierungszyklus korrigiert werden. Die statische Validierung stellt eine Grundlage der in Abschnitt 5.3.2 vorgestellten dynamischen Validierung von Prozeßspezifikationen dar.

Beispielsweise ist eine relevante Systemkonfiguration des in Anhang C.2 spezifizierten Fernsehgeräts durch die Zustände *Television* und *Power_Off* gegeben (vgl. Abbildung 4.7 auf Seite 55). Das Anlegen einer Ausgangskonfiguration zur Verhaltensvalidierung, welche die Zustände *Television*, *Power_Off* und *Power_On* umfaßt, führt zu einem Fehler, da diese drei Zustände nicht in einer maximal konsistenten Beziehung zueinander stehen (vgl. Abschnitt 4.4.1).

Die statische Validierung wird innerhalb der in Abschnitt 5.4 dargestellten Modellierungs- und Validierungsumgebung bei der Anlage einer Ausgangskonfiguration durch den Anwender und beim Übergang von einer Konfiguration in eine Folgekonfiguration im Rahmen der in Abschnitt 5.3.2 vorgestellten dynamischen Validierung durchgeführt.

5.3.2 Dynamische Validierung

Bei der dynamischen Validierung ist die Zielsetzung, festzustellen, ob das spezifizierte Systemverhalten den Erwartungen der Anwender entspricht. Anhand von Szenarios typischer Interaktionssequenzen (vgl. Abschnitt 2.2.2) können Prozeßspezifikationen auf ihre operationale Validität untersucht werden. Vorgenommen wird die dynamische Validierung entweder auf der Grundlage von spezifizierten Ereignispfaddiagrammen oder, wie in Abschnitt 5.2.2 dargestellt, durch das interaktive Auslösen von Ereignissen durch den Anwender selbst.

Ein Beispiel eines Validierungsablaufs der in Anhang C.2 spezifizierten Fernsehersteuerung ist in Tabelle 5.2 dargestellt.

Konfiguration	**Ereignis**	**Folgekonfiguration**
Television, Power_Off	*power*	*Television, Power_On, Standby*
Television, Power_On, Standby	*sound_on*	*Television, Power_On, Standby*
Television, Power_On, Standby	*power*	*Television, Power_Off*

Tabelle 5.2: Dynamischer Validierungslauf

Im Rahmen der in Abschnitt 5.4 dargestellten Tycoon Modellierungs- und Validierungsumgebung wird eine Ausgangskonfiguration, auf deren Grundlage die Validierung des spezifizierten Systemverhaltens vorgenommen werden soll, wie folgt angelegt:

let *stateSet = emptySet(:television.State)*
set.insert(stateSet tv)
set.insert(stateSet powerOff)

television.setConfiguration(stateSet)

Innerhalb des in Anhang C.2 spezifizierten Tycoon Statecharts *Television* werden hierbei die Zustände *Television* (an den Tycoon Bezeichner *tv* gebunden) und *Power_Off* (an den Tycoon Bezeichner *powerOff* gebunden) als Zustände einer Konfiguration ausgezeichnet (vgl. Abbildung 4.7).

Daraufhin werden die externen Ereignisse, für die die Prozeßspezifikation des zu entwerfenden Systems validiert werden soll, in eine Liste eingefügt, welche anschließend im Rahmen der Validierung abgearbeitet wird:

let *eventList = list.new(:television.Event)*
let *eventList1 = list.cons(power eventList)*
let *eventList2 = list.cons(soundOn eventList1)*
let *eventList3 = list.cons(power eventList2)*

Die TL Ereignisliste *eventList3* umfaßt in diesem Fall die externen Ereignisse *power* (gebunden an den Tycoon Bezeichner *power*) und *sound_on* (an den Tycoon Bezeichner *soundOn* gebunden).

Ausgehend von der oben spezifizierten Ausgangskonfiguration innerhalb des *Television* Tycoon Statecharts und der Ereignisliste *eventList3* wird die Tycoon Spezifikation des dynamischen Systemverhaltens (an den Tycoon Bezeichner *television* gebunden) ausgeführt:

television.simulate(eventList3)

Das Tycoon System gibt für jedes externe Ereignis die innerhalb der Prozeßspezifikation vorgenommen Zustandsübergänge (vgl. *Micro Steps* und *Macro Steps*) aus. Nach jeweils einem *Macro Step* wird die stabile Folgekonfiguration angezeigt, so daß der Anwender diese gegen seine Erwartung untersuchen kann, um gegebenenfalls Änderungen in der Prozeßspezifikation des zu erstellenden Systems vorzunehmen.

Ersichtlich wird in Tabelle 5.2, daß ein in der Konfiguration der Zustände *Television*, *Power_On* und *Standby* nicht durchführbares Ereignis *sound_on* nicht zu einem Zustandsübergang und einer damit verbundenen Konfigurationsänderung führt.

In Anhang C.3 ist exemplarisch ein Validierungsvorgang für den in Abbildung 2.4 spezifizierten Ereignispfad dargestellt. Berücksichtigt ist dabei ein Ausschnitt des kompletten Ereignispfads. Ereignisse zwischen den Klassen *Remote-Control* und *Video Recorder* und den Klassen *Video Recorder* und *Television* entfallen.

Auf ähnliche Weise wird eine interaktive Simulation (vgl. Anbindung des externen Dienstes StarView an den Tycoon Validierungsprototypen in Abschnitt 5.2.2) durchgeführt. Aus einem externen StarView Ereignis wird eine Ereignisliste generiert, welche ein einziges, mit diesem StarView Ereignis assoziiertes Ereignis der zu validierenden Prozeßspezifikation umfaßt. Dieses externe Ereignis kann daraufhin einen *Macro Step* innerhalb der Spezifikation des Systemverhaltens auslösen. Anhand der Ausgabe der ausgelösten Zustandsübergänge und der eingenommenen Folgekonfiguration hat der Anwender zu entscheiden, ob das spezifizierte Systemverhalten seiner Erwartung entspricht.

5.4 Tycoon Modellierungs- und Validierungsumgebung

Wie in den vorangegangenen Abschnitten dargestellt, erfolgt der exemplarische Aufbau und die Implementierung der vorgestellten Umgebung für die Modellierung und Validierung dynamischen Systemverhaltens auf der Basistechnologie des Tycoon Systems. In seinem sprachlichen und architektonischen Rahmen wird auf die benötigten generischen, externen Dienste der StP Umgebung und der grafischen Benutzeroberfläche StarView über Tycoon Bibliotheken zugegriffen. Die in Kapitel 4 vorgestellte Modellierungsfunktionalität und die in diesem Kapitel dargestellten Validierungsoperationen werden in TL, der Programmiersprache des Tycoon Systems, realisiert und ebenfalls als Tycoon Bibliotheken dem Tycoon Anwendungsmodellierer zur Verfügung gestellt (vgl. Abschnitt 5.4.2).

Die in Abschnitt 5.4.1 vorgestellte Umgebung für die Modellierung und Validierung dynamischen Systemverhaltens beinhaltet die folgenden Komponenten:

- StP/OMT als methodenbasiertes CASE Werkzeug (*Front End* der Umgebung) zur grafischen Entwicklung von Verhaltensspezifikationen ereignisgesteuerter Systeme innerhalb des StP/OMT dynamischen Modells. Desweiteren wird StP/OMT zur Erstellung von Ereignisfluß- und Ereignispfaddiagrammen im Rahmen der Ereignisidentifikation typischer Interaktionssequenzen zwischen der Systemumgebung und dem zu erstellenden Softwaresystem verwendet.
- StP Repository zur persistenten Ablage der StP Diagrammobjekte eines spezifizierten dynamischen Modells.

- Tycoon System als Implementierungstechnologie für die Modellierung und Validierung dynamischen Systemverhaltens ereignisgesteuerter Systeme mit TL. Darüberhinaus realisiert es den Zugriff auf die StP Modellierungsobjekte im StP Repository und das modellbasierte *Prototyping* auf der Basis eines dynamischen StP/OMT Modells inklusive der Anbindung externer Dienste.
- StP/OMT als mögliches Visualisierungswerkzeug (*Back End* der Umgebung) für innerhalb des Tycoon Systems textuell entworfene und im StP Repository abgelegte Verhaltensspezifikationen zu entwerfender Systeme.

5.4.1 Umgebungsumfang

Der Umfang der im Rahmen dieser Arbeit exemplarisch aufgebauten und implementierten Umgebung für die Modellierung und Validierung dynamischen Systemverhaltens und die darin vorgenommene Aufgabenverteilung auf die eingesetzten Werkzeuge ist in Abbildung 5.6 dargestellt. Dabei umfaßt die Umgebung die in Abschnitt 5.4 wiedergegebenen Komponenten.

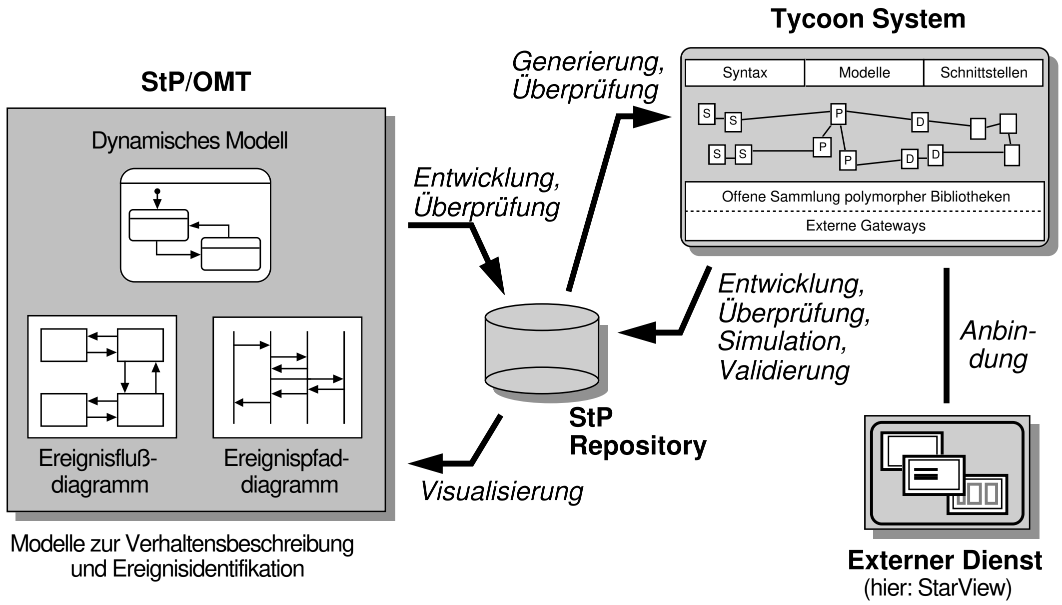


Abbildung 5.6: Modellierungs- und Validierungsumgebung

Mit dem CASE Werkzeug StP/OMT werden Modelle zur Verhaltensbeschreibung und Ereignisidentifikation ereignisgesteuerter Systeme entwickelt. Diese Diagramme des dynamischen Modells, des Ereignisflusses und der Ereignispfade werden auf syntaktische Korrektheit überprüft, bevor die einzelnen Diagrammkomponenten im StP Repository abgelegt werden.

Im Rahmen des in Abschnitt 5.2.1 vorgestellten modellbasierten *Prototyping* werden die StP Repositoryobjekte eines zu validierenden dynamischen Modells identifiziert. Auf sie wird vom Tycoon System lesend zugegriffen, um eine adäquate Tycoon Repräsentation des zu validierenden StP/OMT dynamischen Modells automatisch zu generieren. Hierbei gewährleisten sowohl die Typisierungseigenschaften des Tycoon Systems als auch explizit implementierte Operationen eine Überprüfung der dynamischen Modelle bei der Generierung eines ausführbaren Tycoon Validierungsprototypen. An diesen Tycoon Prototypen können externe Dienste angebunden werden, um die in Abschnitt 5.3 erläuterten Validierungsoperationen durchzuführen.

Zusätzlich zu dem erläuterten modellbasierten *Prototyping* kann im Rahmen der Tycoon Modellierungsumgebung *StatechartEnv* eine textuelle Repräsentation des Systemverhaltens eines zu entwerfenden Systems entwickelt werden. Auch hierbei leisten die Typisierungseigenschaften des Tycoon Systems und explizit implementierte Operationen eine Überprüfung der Modelle.

Mit Hilfe der Anbindung externer Dienste ist sowohl eine interaktive Validierung des spezifizierten Systemverhaltens als auch eine Validierung zum Nachweis der Durchführbarkeit spezifizierter Szenarios möglich (in Abbildung 5.6 auch als Übereinstimmung bezeichnet).

Die Tycoon Repräsentation eines dynamischen Modells kann auf das StP Repository abgebildet werden. Das Werkzeug StP/OMT kann daraufhin zur grafischen Visualisierung des mit TL spezifizierten Systemverhaltens ausgehend von den im Repository abgelegten Modellierungsobjekten benutzt werden. Die dafür erforderliche Abbildung der Tycoon Repräsentation eines dynamischen Modells auf die entsprechenden StP Repositoryobjekte ist allerdings kein Untersuchungspunkt dieser Arbeit.

5.4.2 Umgebungsarchitektur

Schematisch dargestellt entspricht die in Abschnitt 5.4.1 vorgestellte Tycoon Umgebung zur Modellierung und Validierung dynamischen Systemverhaltens der in Abbildung 5.7 gezeigten Schichtenarchitektur.

In der Schicht der Anwendungsmodellierung erstellt ein Tycoon Anwendungsmodellierer zwecks Validierung des spezifizierten Systemverhaltens ein dynamisches Modell eines Systems. Dieses Modell kann, wie in Kapitel 4 gezeigt, in einer textuellen Repräsentation programmiert werden oder automatisch aus einem grafischen StP/OMT Modell generiert werden (vgl. Abschnitt 5.2.1).

Eine Modellierung und Validierung dynamischen Systemverhaltens unabhängig von der StP Umgebung anhand einer textuellen Tycoon Repräsentation wird durch die in Anhang B.1.1 dargestellten Funktionen des Moduls *Statechart* der Tycoon Bibliothek *StatechartEnv* realisiert. Die Anbindung eines externen Dienstes erfolgt durch den Import der entsprechenden Funktionen aus den innerhalb des Tycoon Systems zur Verfügung gestellten Bibliotheken.

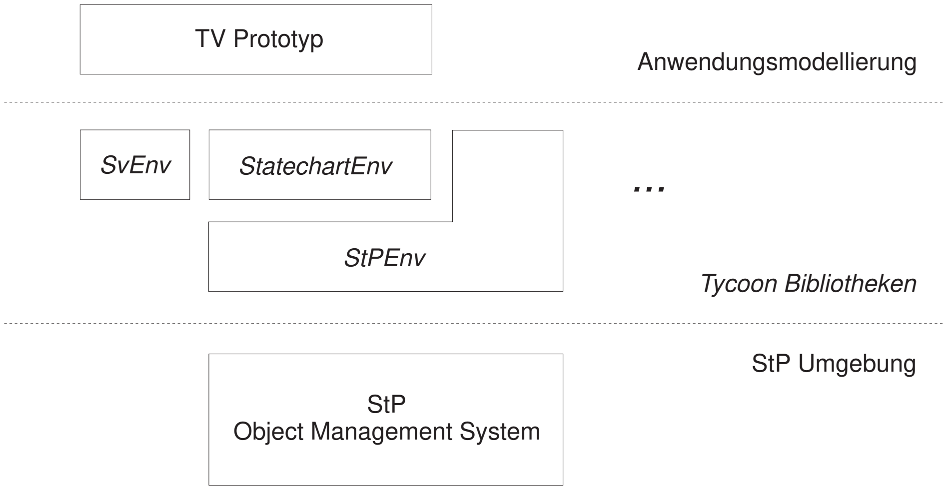


Abbildung 5.7: Umgebungsarchitektur

Als externer Dienst kann z.B. die für die interaktive Validierung benötigte StarView Funktionalität aus der Bibliothek *SvEnv* importiert werden. Aber auch sonstige im Tycoon System verfügbare, externen Dienste können an die Tycoon Repräsentation eines dynamischen Modells angebunden werden.

Die Generierung des Tycoon Validierungsprototypen *TV* (vgl. Abschnitt 5.2.1) basiert auf dem in der Tycoon Bibliothek *StatechartEnv* enthaltenen Modul *StP2Tycoon* (vgl. Anhang B.1.2), das unter Verwendung der Funktionalität des Moduls *Statechart* und der Module der Bibliothek *StPEnv* einen Validierungsprototypen in Tycoon Repräsentation aus einem StP/OMT dynamischen Modell erstellt. Die Tycoon Bibliothek *StPEnv* stellt, wie in Abschnitt 3.3 dargestellt, den Zugriff auf die Funktionalität des StP Object Management Systems innerhalb der StP Umgebung zur Verfügung, welche einen Zugriff auf die im StP Repository abgelegten Modellierungsdaten realisiert.

6. Zusammenfassung und Ausblick

Ausgangspunkt dieser Arbeit ist die Betrachtung des Softwareentwicklungsprozesses, in dem die Softwareentwicklung den Einsatz leistungsfähiger und integrierter Entwicklungswerkzeuge bedingt, um die in Abschnitt 1.1 dargestellten Produktivitätsdeterminanten eines Softwareentwicklungsprojekts zu erfüllen. Zu diesem Zweck wird im Rahmen dieser Arbeit ein methodisches Vorgehen zur Anwendungsentwicklung und dessen Werkzeugunterstützung zur computerbasierten Systemspezifikation betrachtet. Der Schwerpunkt liegt hierbei auf der Validierung des dynamischen Systemverhaltens.

Realisiert wird die methodenorientierte Unterstützung des Softwareentwicklungsprozesses durch die in Abschnitt 2.2 dargestellte objektorientierte Anwendungsentwicklungsmethode *Object Modeling Technique* (OMT). Im Rahmen der Spezifikation dynamischen Systemverhaltens stellt OMT innerhalb der Analyse ein dynamisches Modell (vgl. Abschnitt 2.2.2) dem Anwendungsentwickler zur Verfügung. Desweiteren bietet OMT die Möglichkeit, identifizierte Ereignisse von der Systemumgebung an das zu erstellende System im Rahmen der Ereignisidentifikation grafisch zu spezifizieren. Desweiteren können in OMT Ereignispfaddiagramme zur Darstellung von Szenarios angelegt werden. Diese Diagramme stellen typische Interaktionssequenzen von Ereignissen zwischen Objektklassen eines Systems zeitabhängig dar.

Implementiert wird die Methode OMT durch das in Abschnitt 2.3 dargestellte CASE Werkzeug StP/OMT, welches in die Software through Pictures CASE Umgebung integriert wird. Allerdings bietet dieses computerbasierte Entwicklungswerkzeug keine Validierungsoperationen, um schon in der Analysephase des Softwarelebenszyklus das spezifizierte Systemverhalten zu validieren. Mit Hilfe der Validierung sollen nachträgliche Konzeptänderungen und kostenintensive Fehlerbehebungen in späteren Projektphasen, z.B. der Implementierung, vermieden werden.

Zur Unterstützung des Softwareentwicklungsprozesses wird in dieser Arbeit eine Umgebung zur Modellierung und Validierung dynamischen Systemverhaltens vorgestellt, die die folgenden Komponenten umfaßt:

- Das methodenbasierte CASE Werkzeug StP/OMT dient zur grafischen Entwicklung von Verhaltensspezifikationen ereignisgesteuerter Systeme innerhalb des StP/OMT dynamischen Modells. Desweiteren wird es zur Erstellung von Ereignisfluß- und Ereignispfaddiagrammen im Rahmen der Ereignisidentifikation typischer Interaktionssequenzen zwischen der Systemumgebung und dem zu erstellenden Softwaresystem verwendet.

- Die persistente Ablage der StP Diagrammobjekte eines spezifizierten dynamischen Modells erfolgt im StP Repository.
- Das Tycoon System dient als Implementierungstechnologie für die Modellierung und Validierung dynamischen Systemverhaltens ereignisgesteuerter Systeme mit TL. Darüberhinaus realisiert es den Zugriff auf die StP Modellierungsobjekte im StP Repository und das modellbasierte *Prototyping* auf der Basis eines dynamischen Modells inklusive der Anbindung externer Dienste.

Diese Modellierungs- und Validierungsumgebung ist in dieser Arbeit exemplarisch aufgebaut (vgl. Abschnitt 5.4) und mit Hilfe des Tycoon Systems und seiner Programmiersprache TL implementiert. Das Tycoon System bietet als integrierte Programmierumgebung, wie in Abschnitt 1.2 dargestellt einen geeigneten sprachlichen und architektonischen Rahmen zur Definition und Integration generischer Dienste in einer offenen Systemumgebung.

Als Basistechnologie zur Implementierung erfüllt das Tycoon System die folgenden Anforderungen der in dieser Arbeit erstellten Modellierungs- und Validierungsumgebung:

- Die in Abschnitt 1.2 dargestellte Offenheit des Tycoon Systems zur Integration externer Dienste ermöglicht eine Integration der StP Object Management Funktionalität, auf die bei der Generierung eines StP/OMT modellbasierten Prototypen eines dynamischen StP/OMT Modells zurückgegriffen wird. Desweiteren können sonstige externe Dienste, wie z.B. die grafische Benutzeroberfläche StarView, als ereignisauslösender oder -aufnehmender Dienst an die zu validierende Tycoon Verhaltensspezifikation eines zu erstellenden Systems angebunden werden.
- Die strikte Typisierung von TL leistet ihren Beitrag unter anderem bei der Modellierung der in Kapitel 4 vorgestellten Tycoon Statecharts, da sie zusichert, daß nur passende Statechartelemente den entsprechenden Funktionen zur Generierung eines Tycoon Statecharts als Parameter übergeben werden (vgl. Abschnitt 4.2 und 4.3).

  Die Verwendung von Funktionen höherer Ordnung in TL ermöglicht die Anbindung von TL Funktionen an Aktionen eines Tycoon Statecharts (vgl. Abschnitt 4.3.2).
- Die Nebenläufigkeitseigenschaften des Tycoon Systems werden ausgenutzt, um separate Kontrollflüsse innerhalb eines Tycoon Validierungsprototypen und eines angebundenen externen Dienstes zu erzielen (vgl. Abschnitt 5.2.2).
- Im Rahmen der Reflektion innerhalb des Tycoon Systems können im StP/OMT Tabelleneditor Tycoon Kodefragmente abgelegt werden, welche bei der Generierung eines

Tycoon Validierungsprototypen an den entsprechenden Stellen zum Generierungszeitpunkt übersetzt und eingebunden werden (vgl. Abschnitt 5.2.1).

- Die Persistenzeigenschaften des Tycoon Systems werden in dieser Arbeit nicht ausgenutzt. Es wäre beispielsweise möglich, einen sich in einer stabilen Konfiguration befindenden Tycoon Statechart im Objektspeicher des Tycoon Systems abzulegen und später für die weitere Validierung zu reaktivieren.

Die Validierungskomponente der im Rahmen dieser Arbeit exemplarisch aufgebauten und implementierten Modellierungs- und Validierungsumgebung kann in die OMT Methode zur Unterstützung der Anwendungserstellung integriert werden. Auf diese Weise können Verhaltensspezifikationen zu erstellender Systeme bereits in der Analysephase gegen die Erwartungen des Anwenders untersucht werden.

Eine Erweiterung der vorliegenden Umgebung zur Modellierung und Validierung dynamischen Systemverhaltens kann erreicht werden, indem valide Ereignispfade eines dynamischen OMT Modells nach der in Abschnitt 5.3 dargestellten dynamischen Validierung in einer Datenbank abgelegt werden. Bei einer Änderung der Verhaltensspezifikation kann diese anhand der in der Datenbank abgelegten, validen Szenarios automatisch validiert werden, so daß eine Verkürzung des in Abschnitt 5.1 erläuterten Validierungszyklus eintritt.

Desweiteren kann die Modellierungs- und Validierungsumgebung zur Spezifikation von Mobilität und Verteilungsaspekten in kooperierenden, persistenten Objektsystemen benutzt werden [Schröder 97]. Eine modellbasierte Generierung entsprechender Prototypen kann ebenfalls erfolgen.

Im Rahmen einer Erweiterung des CASE Werkzeugs StP/OMT um eine Workflownotation [Stamer 97] kann die in dieser Arbeit dargestellte Modellierungs- und Validierungsumgebung auf die Simulation von Workflowmodellen ausgedehnt werden.

A. Anbindung der StP API Funktionen an das Tycoon System

A.1 Generierungsskript für dynamische StP Bibliothek lipstp.so

```
ld -G -o ${TYCOON_ROOT}/lib/${TYCOON_HOST}/libstp.so
.../StP/libsparc/libideoms.so .../StP/libsparc/libideparse.so
.../StP/libsparc/libideeditor.so .../StP/libsparc/libidetools.so
.../StP/libsparc/libsybdb.so /usr/lib/libm.so
.../StP/libsparc/libidestub.so /usr/lib/libnsl.so
```

A.2 Pfade zu den StP Systemdateien

```
stp_file_path=.../StP/templates/IDE/omt_booch:
.../StP/templates/IDE/booch:.../StP/templates/IDE/oo:
.../StP/templates/IDE/omtcc:.../StP/templates/IDE/t:
.../StP/templates/IDE/ct:.../StP/templates/IDE
```

A.3 StP Systemumgebungsvariablen

```
LD_LIBRARY_PATH=.../StP/libsparc:/usr/openwin/lib:/opt/sv/lib
PATH=.../StP/binsparc:.../StP/sybase_10.0.1/sparc/bin:
     /usr/ccs/bin:/opt/SUNWspro/SW3.0.1/bin
SHLVL=3 IDE_PRODUCT=omt ToolInfo=.../StP/ToolInfo/IDE
SYBASE=.../StP/sybase_10.0.1/sparc DSQUERY=STP_SERVER
SYS_TEN=true STP_SERVER_PASSWD=welcome
```

A.4 Bindung und Verwendung der StP API Funktionalität

```
Let T = omsRep.NodeHandle
let lib = runtimeCore.dynamicLibraryName("stpenv" "libstp")
let handleNil = word.HandleNil(:omsRep.Node)
let error = exception "OMS_ERROR" end

let findCall = bind(:Fun(:String :omsApplicationType.T :omsObjectId.T :String)
                    :T lib "oms_node_find" "siwsw")

let find(rep :omsRepository.T  name :String  type :omsApplicationType.T
        scopeNodeId :omsObjectId.T(T)  signature :String) :T =
 begin
  omsRepository.check(rep)
  let omsNodeTp = findCall(rep name type scopeNodeId signature)
  if word.handleEqual(omsNodeTp handleNil) then
   raise error end
  else
   omsNodeTp
  end
 end
```

B. Tycoon Schnittstellen

B.1 Modellierungs- und Validierungsumgebung StatechartEnv

B.1.1 Modul Statechart

```
interface Statechart
(*
  System:  StatechartEnv
  File:    Statechart.ti
  Author:  Oliver Nietsch
  Date:    23-SEP-1996
  Purpose: Statecharts (see: D. Harel. "Statecharts: A Visual Formalism for
                         Complex Systems". In: Science of Computer Programming,
                         Nr. 8, Amsterdam, S. 231-274, 1987)
*)
import
 list  set

export
 Let Chart <:Ok =
  Tuple

    Event <:Ok
    Let Events = list.T(Event)
    Action <:Ok
    Let Actions = set.T(Action)
    Activity <:Ok
    Let Activities = set.T(Activity)
    State <:Ok
    Let States = set.T(State)
    Transition <:Ok
    Let Transitions = set.T(Transition)
```

```
(* Events and Actions *)

createPrimitiveEvent(trigger :String) :Event

createAction(actionLabel :String  triggeredEvent :Event
  actionPart :Fun() :Ok) :Action

setActionPart(action :Action  actionPart :Fun() :Ok) :Ok

(* States *)

createBasicState(stateLabel :String  actionsWhenEntering :Actions
  actionsWhenLeaving :Actions  throughoutActivities :Activities) :State

createCompositeAndState(stateLabel :String  actionsWhenEntering :Actions
  actionsWhenLeaving :Actions  throughoutActivities :Activities
  children :States) :State

createCompositeOrState(stateLabel :String  actionsWhenEntering :Actions
  actionsWhenLeaving :Actions  throughoutActivities :Activities
  children :States  defaultState :State) :State

createRootState(stateLabel :String  actionsWhenEntering :Actions
  actionsWhenLeaving :Actions  throughoutActivities :Activities
  children :States  defaultState :State) :State

(* Transitions *)

createNormalTransition(source :State  target :State  trigger :Events) :Transition

createConditionalTransition(source :State  target :State  trigger :Events
  conditions :States) :Transition

createTransitionWithActions(source :State  target :State  trigger :Events
  actions :Actions) :Transition

createConditionalTransitionWithActions(source :State  target :State
  trigger :Events  conditions :States  actions :Actions) :Transition

(* Configuration *)

setConfiguration(states :States) :Ok
```

```
    printConfiguration() :Ok

    (* Statechart Execution *)

    enableTransitions(transition :Transition  input :Events) :Ok

    simulate(input :Events) :Ok

   end (* Tuple *)


  (* New statechart with functions above *)

  newChart() :Chart


end; (* Statechart *)
```

B.1.2 Modul StP2Tycoon

```
interface StP2Tycoon
(*
   System:  StatechartEnv
   File:    StP2Tycoon.ti
   Author:  Oliver Nietsch
   Date:    14-DEZ-1996
   Purpose: Creation of Tycoon Validation Prototype of Rootstate
            StP Dynamic Model from selected StP Project Directory and StP System
   Runtime Condition: libstp.so in $TYCOON_ROOT/lib/$TYCOON_HOST
   Used Tycoon Libraries: $TYCOON_ROOT/src/stpenv
*)
import
  statechart

export
  generateStatechart(projdir, system, rootState :String) :statechart.Chart


end; (* StP2Tycoon *)
```

B.2 StP Repository Gateway StPEnv

B.2.1 Module mit spezieller Tycoon Funktionalität

Modul	Beschreibung
omsError	Spezifikation der auszuführenden Fehlerroutine beim Auftreten eines StP OMS Fehlers.
omsQuery	Kapselung von StP OMS Anfragen.
omsRep	Enthält die Deklaration der wichtigsten Implementationsdatentypen.

B.2.2 Module mit spezieller StP PDM Datentyp Funktionalität

Modul	Beschreibung
omsGeneric	Generische Hilfsfunktionen, Repositorymanagementfunktionen, Operationen zum Auffinden von Repositoryobjekten und Attributzugriffs- und Zuweisungsfunktionen für beliebige/alle PDM Datentypen.
oms<pdm_typ>	Operationen aus *omsGeneric* und spezielle Funktionen zum Erstellen und Auffinden von Repositoryobjekten und Navigationsoperationen.

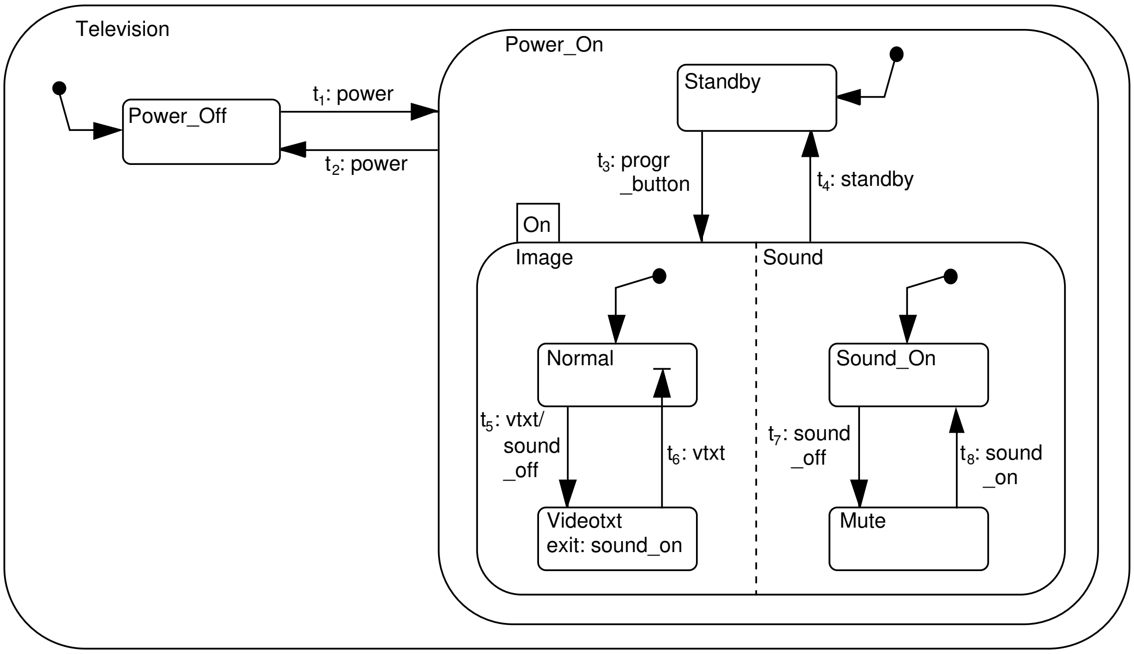
B.2.3 Module mit allgemeiner StP PDM Datentyp Funktionalität

Modul	**Beschreibung**
omsBigCollection	Generische Operationen zum Generieren und Manipulieren von *Big Collection* Objekten beliebiger/aller PDM Datentypen.
omsCollection	Generische Operationen zum Generieren und Manipulieren von *Collection* Objekten beliebiger/aller PDM Datentypen.
omsApplicationType	Operationen entsprechend des StP API Datentyps `oms_app_tp` zur Abbildung von Anwendungsdatentypen auf PDM Datentypen.
omsIdList	Operationen entsprechend des StP API Datentyps `oms_idlist_tp` zum Verwalten von ID Listen.
omsPDMType	Operationen entsprechend des StP API Datentyps `oms_pdm_type_tp` zur Verwaltung der internen Darstellung der einzelnen PDM Datentypen.
omsObjectId	Operationen entsprechend des StP API Datentyps `oms_object_id_tp` zum Erstellen einer individuellen Objekt ID für Objekte innerhalb des StP Repositorys.
omsRepository	Operationen entsprechend des StP API Datentyps `oms_repos_tp` zum Initialisieren der StP Systemvariablen, Öffnen, Verwalten und Schließen des StP Repositorys.
omsStatus	Operationen entsprechend des StP API Datentyps `oms_status_tp` zum Überprüfen ob eine API Operation erfolgreich ausgeführt werden konnte.
omsTime	Operationen entsprechend des StP API Datentyps `oms_time_tp` zur Operation auf Datums- und Zeitrepräsentationen.
omsTransaction	Operationen entsprechend des StP API Datentyps `oms_txn_tp` zum Verwalten von StP Repository Transaktionen.

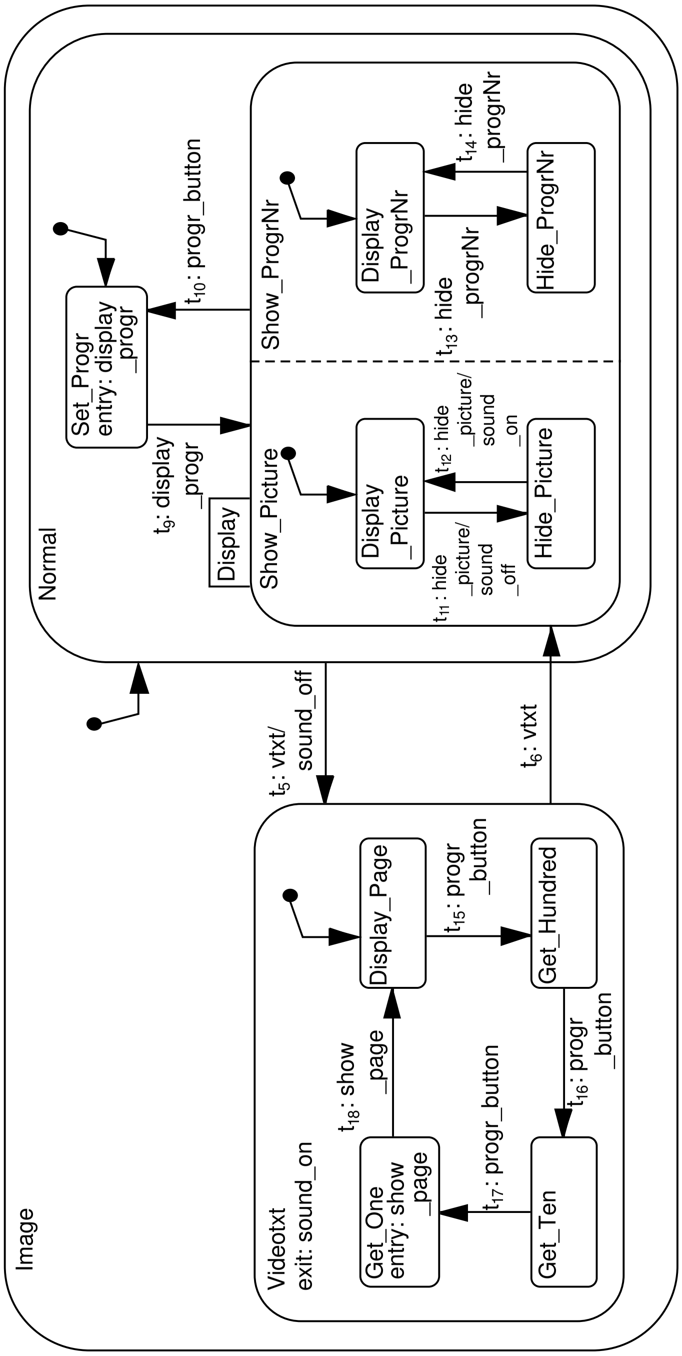
C. Anwendungsbeispiel Television

C.1 Statecharts

C.1.1 Zustand Television



C.1.2 Verfeinerung Zustand Image



C.2 TL Repräsentation

```
module statechartTelevision
(*
  System: statechartDemo
  File: statechartTelevision.tm
  Author: Oliver Nietsch
  Date: 07-OCT-1996
  Purpose: Creation of Television Statechart Elements
*)
import

 list set statechart

export

 let television = statechart.newChart()

 Let Event = television.Event
 Let Events = list.T(Event)
 Let Action = television.Action

 (* Auxiliary Functions *)
 let emptySet(A <:Ok) :set.T(A) = set.new(:A fun(element1, element2 :A) :Bool
   element1 == element2)

 (* Triggers *)
 let triggerPower = television.createPrimitiveEvent("power")
 let triggerProgrButton = television.createPrimitiveEvent("progr_button")
 let triggerStandby = television.createPrimitiveEvent("standby")
 let triggerVtxt = television.createPrimitiveEvent("vtxt")
 let triggerSoundOff = television.createPrimitiveEvent("sound_off")
 let triggerSoundOn = television.createPrimitiveEvent("sound_on")
 let triggerDisplayProgr = television.createPrimitiveEvent("display_progr")
 let triggerHidePicture = television.createPrimitiveEvent("hide_picture")
 let triggerHideProgrNr = television.createPrimitiveEvent("hide_progrNr")
 let triggerShowPage = television.createPrimitiveEvent("show_page")

 (* Actions *)
 let actionSoundOff = television.createAction("sound_off" triggerSoundOff fun() :Ok ok)
 let actionDisplayProgr = television.createAction("display_progr"
   triggerDisplayProgr fun() :Ok ok)
```

```
let actionSoundOn = television.createAction("sound_on" triggerSoundOn fun() :Ok ok)
let actionShowPage = television.createAction("show_page"
  triggerShowPage fun() :Ok ok)

(* States *)
(* — Videotxt substate — *)
let displayPage = television.createBasicState("Display_Page"
  emptySet(:television.Action) emptySet(:television.Action)
  emptySet(:television.Activity))
let getHundred = television.createBasicState("Get_Hundred"
  emptySet(:television.Action) emptySet(:television.Action)
  emptySet(:television.Activity))
let getTen = television.createBasicState("Get_Ten" emptySet(:television.Action)
  emptySet(:television.Action) emptySet(:television.Activity))
let actionSet = emptySet(:television.Action)
set.insert(actionSet actionShowPage)
let getOne = television.createBasicState("Get_One" actionSet
  emptySet(:television.Action) emptySet(:television.Activity))
let stateSet = emptySet(:television.State)
set.insert(stateSet displayPage)
set.insert(stateSet getHundred)
set.insert(stateSet getTen)
set.insert(stateSet getOne)
let actionSet = emptySet(:television.Action)
set.insert(actionSet actionSoundOn)
let videotxt = television.createCompositeOrState("Videotxt"
  emptySet(:television.Action) actionSet
  emptySet(:television.Activity) stateSet displayPage)

(* — Show_ProgrNr substate — *)
let displayProgrNr = television.createBasicState("Display_ProgrNr"
  emptySet(:television.Action) emptySet(:television.Action)
  emptySet(:television.Activity))
let hideProgrNr = television.createBasicState("Hide_ProgrNr"
  emptySet(:television.Action) emptySet(:television.Action)
  emptySet(:television.Activity))
let stateSet = emptySet(:television.State)
set.insert(stateSet displayProgrNr)
set.insert(stateSet hideProgrNr)
let showProgrNr = television.createCompositeOrState("Show_ProgrNr"
  emptySet(:television.Action) emptySet(:television.Action)
  emptySet(:television.Activity) stateSet displayProgrNr)
```

```
(* — Show_Picture substate — *)
let displayPicture = television.createBasicState("Display_Picture"
  emptySet(:television.Action) emptySet(:television.Action)
  emptySet(:television.Activity))
let hidePicture = television.createBasicState("Hide_Picture" emptySet(:television.Action)
  emptySet(:television.Action) emptySet(:television.Activity))
let stateSet = emptySet(:television.State)
set.insert(stateSet displayPicture)
set.insert(stateSet hidePicture)
let showPicture = television.createCompositeOrState("Show_Picture"
  emptySet(:television.Action) emptySet(:television.Action)
  emptySet(:television.Activity) stateSet displayPicture)

(* — Display substate — *)
let stateSet = emptySet(:television.State)
set.insert(stateSet showPicture)
set.insert(stateSet showProgrNr)
let display = television.createCompositeAndState("Display" emptySet(:television.Action)
  emptySet(:television.Action) emptySet(:television.Activity) stateSet)

(* — Normal substate — *)
let actionSet = emptySet(:television.Action)
set.insert(actionSet actionDisplayProgr)
let setProgr = television.createBasicState("Set_Progr" actionSet
  emptySet(:television.Action) emptySet(:television.Activity))
let stateSet = emptySet(:television.State)
set.insert(stateSet display)
set.insert(stateSet setProgr)
let normal = television.createCompositeOrState("Normal" emptySet(:television.Action)
  emptySet(:television.Action) emptySet(:television.Activity) stateSet setProgr)

(* — Image substate — *)
let stateSet = emptySet(:television.State)
set.insert(stateSet normal)
set.insert(stateSet videotxt)
let image = television.createCompositeOrState("Image" emptySet(:television.Action)
  emptySet(:television.Action) emptySet(:television.Activity) stateSet normal)

(* — Sound substate — *)
let soundOn = television.createBasicState("Sound_On" emptySet(:television.Action)
  emptySet(:television.Action) emptySet(:television.Activity))
```

```
let mute = television.createBasicState("Mute" emptySet(:television.Action)
  emptySet(:television.Action) emptySet(:television.Activity))
let stateSet = emptySet(:television.State)
set.insert(stateSet soundOn)
set.insert(stateSet mute)
let sound = television.createCompositeOrState("Sound" emptySet(:television.Action)
  emptySet(:television.Action) emptySet(:television.Activity) stateSet soundOn)

(* — On substate — *)
let stateSet = emptySet(:television.State)
set.insert(stateSet image)
set.insert(stateSet sound)
let on = television.createCompositeAndState("On" emptySet(:television.Action)
  emptySet(:television.Action) emptySet(:television.Activity) stateSet)

(* — Power_On substate — *)
let standby = television.createBasicState("Standby" emptySet(:television.Action)
  emptySet(:television.Action) emptySet(:television.Activity))
let stateSet = emptySet(:television.State)
set.insert(stateSet standby)
set.insert(stateSet on)
let powerOn = television.createCompositeOrState("Power_On"
  emptySet(:television.Action) emptySet(:television.Action)
  emptySet(:television.Activity) stateSet standby)

(* — Television superstate — *)
let powerOff = television.createBasicState("Power_Off" emptySet(:television.Action)
  emptySet(:television.Action) emptySet(:television.Activity))
let stateSet = emptySet(:television.State)
set.insert(stateSet powerOff)
set.insert(stateSet powerOn)
let tv = television.createRootState("Television" emptySet(:television.Action)
  emptySet(:television.Action) emptySet(:television.Activity) stateSet powerOff)

(* Transitions *)
let trigger1 = list.cons(triggerPower list.new(:television.Event))
let t1 = television.createNormalTransition(powerOff powerOn trigger1)
let t2 = television.createNormalTransition(powerOn powerOff trigger1)
let trigger2 = list.cons(triggerProgrButton list.new(:television.Event))
let t3 = television.createNormalTransition(standby on trigger2)
let t10 = television.createNormalTransition(display setProgr trigger2)
let t15 = television.createNormalTransition(displayPage getHundred trigger2)
```

let *t16 = television.createNormalTransition(getHundred getTen trigger2)*
let *t17 = television.createNormalTransition(getTen getOne trigger2)*
let *trigger3 = list.cons(triggerStandby list.new(:television.Event))*
let *t4 = television.createNormalTransition(on standby trigger3)*
let *trigger4 = list.cons(triggerVtxt list.new(:television.Event))*
let *action1 = emptySet(:television.Action)*
set.insert(:television.Action action1 actionSoundOff)
let *t5 = television.createTransitionWithActions(normal videotxt trigger4 action1)*
let *t6 = television.createNormalTransition(videotxt display trigger4)*
let *trigger5 = list.cons(triggerSoundOff list.new(:television.Event))*
let *t7 = television.createNormalTransition(soundOn mute trigger5)*
let *trigger6 = list.cons(triggerSoundOn list.new(:television.Event))*
let *t8 = television.createNormalTransition(mute soundOn trigger6)*
let *trigger7 = list.cons(triggerDisplayProgr list.new(:television.Event))*
let *t9 = television.createNormalTransition(setProgr display trigger7)*
let *trigger8 = list.cons(triggerHidePicture list.new(:television.Event))*
let *t11 = television.createTransitionWithActions(displayPicture*
hidePicture trigger8 action1)
let *action2 = emptySet(:television.Action)*
set.insert(:television.Action action2 actionSoundOn)
let *t12 = television.createTransitionWithActions(hidePicture*
displayPicture trigger8 action2)
let *trigger9 = list.cons(triggerHideProgrNr list.new(:television.Event))*
let *t13 = television.createNormalTransition(displayProgrNr hideProgrNr trigger9)*
let *t14 = television.createNormalTransition(hideProgrNr displayProgrNr trigger9)*
let *trigger10 = list.cons(triggerShowPage list.new(:television.Event))*
let *t18 = television.createNormalTransition(getOne displayPage trigger10)*

end*; (* statechartTelevision *)*

C.3 Szenariobasierter Validierungslauf

Konfiguration	**Ereignis**	**Folgekonfiguration**
Television, Power_On, Standby	*progr_button*	*Television, Power_On, On, Image, Normal, Display, Show_Picture, Display_Picture, Show_ProgrNr, Display_ProgrNr, Sound, Sound_On*
⋮	⋮	⋮
Television, Power_On, On, Image, Normal, Display, Show_Picture, Display_Picture, Show_ProgrNr, Display_ProgrNr, Sound, Sound_On	*vtxt*	*Television, Power_On, On, Image, Videotxt, Display_Page, Sound, Sound_Off*
⋮	⋮	⋮
Television, Power_On, On, Image, Videotxt, Display_Page, Sound, Sound_Off	*vtxt*	*Television, Power_On, On, Image, Normal, Display, Show_Picture, Display_Picture, Show_ProgrNr, Display_ProgrNr, Sound, Sound_Off*
Television, Power_On, On, Image, Normal, Display, Show_Picture, Display_Picture, Show_ProgrNr, Display_ProgrNr, Sound, Sound_Off	*sound_off*	*Television, Power_On, On, Image, Normal, Display, Show_Picture, Display_Picture, Show_ProgrNr, Display_ProgrNr, Sound, Sound_Off*
Television, Power_On, On, Image, Normal, Display, Show_Picture, Display_Picture, Show_ProgrNr, Display_ProgrNr, Sound, Sound_Off	*standby*	*Television, Power_On, Standby*

Literaturverzeichnis

Atkinson et al. 91: M. Atkinson, R. Morrison, und D. Stemple. „Type-safe Linguistic Reflection“. In: *Proceedings of the Third International Workshop on Database Programming Languages, Nafplion, Griechenland. Morgan Kaufmann, San Mateo u.a.*, Seite 357–362, 1991.

Balzert 93: H. Balzert. *CASE — Systeme und Werkzeuge.* Bibliographisches Institut, Mannheim, 1993.

Boehm 79: B.W. Boehm. „Software Engineering: R & D Trends and Defense Needs“. In: *P. Wegner (Hrsg.). Research Directions in Software Technology. MIT Press*, 1979.

Chen 76: P.P.S. Chen. „The Entity-Relationship Model — Toward a Unified View of Data“. In: *ACM Transactions on Database Systems, Jahrg. 1, Nr.1, New York*, Seite 9–36, 1976.

Day 93: N. Day. „A Model Checker for Statecharts“. Technical Report 93-35, Department of Computer Science, University of British Columbia, Vancouver, Kanada, 1993.

Fricke 95: O. Fricke. „Die Strukturierte Analyse auf der Basis höherer Petrinetze“. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, 1995.

Geisler 95: A. Geisler. „Basisdienste zur Gestaltung einer reflektiven grafischen Entwicklungsumgebung für eine persistente Programmiersprache“. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, 1995.

Glinz 93: M. Glinz. „Hierarchische Verhaltensbeschreibung in objektorientierten Systemmodellen — eine Grundlage für modellbasiertes Prototyping“. In: *H. Züllighoven, W. Altmann und E.E. Doberkat (Hrsg.). Requirements Engineering '93: Prototyping. Berichte des German Chapter of the ACM, Band 41. Teubner, Stuttgart*, Seite 175–192, 1993.

Glinz 95: M. Glinz. „An Integrated Formal Model of Scenarios Based on Statecharts“. In: *W. Schäfer und P. Botella (Hrsg.). Software Engineering - ESEC '95. Springer, Berlin u.a.*, Seite 231–271, 1995.

Guttenberg 91: V. Guttenberg. „Entwicklung eines graphischen Monitors zur Validierung zeitdiskreter Simulationsmodelle“. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, 1991.

Hachmann 96: I. Hachmann. „Generische Visualisierungsdienste am Beispiel von StarView und Tycoon“. Studienarbeit, Universität Hamburg, Fachbereich Informatik, 1996.

Harel et al. 90: D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, und M. Trakhtenbrot. „STATEMATE: A Working Environment for the Development of Complex Reactive Systems“. In: *IEEE Transactions on Software Engineering, Jahrg. 16, Nr.4, New York*, Seite 401–413, 1990.

Harel, Pnueli 85: D. Harel und A. Pnueli. „On the Development of Reactive Systems“. In: *K.R. Apt (Hrsg.). Logics and Models of Concurrent Systems, Springer, Berlin u.a*, Seite 477–498, 1985.

Harel 87: D. Harel. „Statecharts: A Visual Formalism for Complex Systems“. In: *Science of Computer Programming, Nr. 8, Amsterdam*, Seite 231–274, 1987.

Henize 84: J. Henize. „Critical Issue in evaluating socio-economic Models“. In: *T.I. Ören et al. (Hrsg.). Simulation and Model-based Methodologies: An Integrative View. Springer, Berlin u.a.*, 1984.

Hopcroft, Ullman 90: J.E. Hopcroft und J.D. Ullman. *Einführung in die Automatentheorie.* Addison-Wesley, Bonn u.a., 1990.

Huizing et al. 88: C. Huizing, R. Gerth, und W.P. de Roever. „Modelling Statecharts Behaviour in a fully abstract Way“. In: *M. Dauchet, M. Nivat (Hrsg.). Lecture Notes in Computer Science 299, Springer, Berlin u.a.*, Seite 271–294, 1988.

Huizing, de Roever 91: C. Huizing und W.P. de Roever. „Introduction to design choices in the semantics of Statecharts“. In: *Information Processing Letters, Jahrg. 37, Nr. 2, Amsterdam*, Seite 205–213, 1991.

IDEOMTModels 93: Interactive Development Environments Inc. (Hrsg.). *Object Modeling Technique — Creating OMT Models. San Francisco, Kalifornien*, 1993.

IDEStPCust 94: Interactive Development Environments Inc. (Hrsg.). *Software through Pictures Core — Customizing StP. San Francisco, Kalifornien*, 1994.

IDEStPIM 93: Interactive Development Environments Inc. (Hrsg.). *Software through Pictures — Product Summary: Information Modeling. San Francisco, Kalifornien*, 1993.

IDEStPOMS 94: Interactive Development Environments Inc. (Hrsg.). *Software through Pictures Core — Object Management System. San Francisco, Kalifornien*, 1994.

IDEStPPC 93: Interactive Development Environments Inc. (Hrsg.). *Software through Pictures — Product Catalog 1993. San Francisco, Kalifornien*, 1993.

IDEStPQRS 94: Interactive Development Environments Inc. (Hrsg.). *Software through Pictures Core — Query and Reporting System. San Francisco, Kalifornien*, 1994.

IDEStPTWP 93: Interactive Development Environments Inc. (Hrsg.). *Software through Pictures Core — Technical White Paper. San Francicsco, Kalifornien*, 1993.

Jessen, Valk 87: E. Jessen und R. Valk. *Rechensysteme: Grundlagen der Modellbildung.* Springer, Berlin u.a., 1987.

Klees, Schmauch 94: F. Klees und C. Schmauch. „Navigator: StP/OMT — Software-Entwicklung nach Rumbaugh". In: *iX, Nr. 4, Hannover*, Seite 62ff., 1994.

Krüger 74: S. Krüger. *Simulation. Grundlagen, Techniken, Anwendungen.* De Gruyter, Berlin, 1974.

Leveson et al. 94: N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, und J.D. Reese. „Requirements Specification for Process-Control Systems". In: *IEEE Transactions on Software Engineering, Jahrg. 20, Nr.9, New York*, Seite 684–707, 1994.

Losavio et al. 94: F. Losavio, A. Matteo, und F. Schlienger. „Metamodels for the Object-Oriented Approaches of Booch and OMT". Rapport de Recherche 926, Laboratoire de Recherche en Informatique, Université de Paris-Sud, Orsay, Frankreich, 1994.

Mathiske et al. 93: B. Mathiske, F. Matthes, und S. Müßig. „The Tycoon System and Library Manual". DBIS Tycoon Report 212-93, Universität Hamburg, Fachbereich Informatik, 1993.

Matthes, Müßig 93: F. Matthes und S. Müßig. „The Tycoon Language TL: An Introduction.". DBIS Tycoon Report 112-93, Universität Hamburg, Fachbereich Informatik, 1993.

Matthes 93: F. Matthes. *Persistente Objektsysteme. Integrierte Datenbankentwicklung und Programmerstellung.* Springer, Berlin, 1993.

Müßig 94: S. Müßig. „Beiträge zur typsicheren generischen Datenvisualisierung". Diplomarbeit, Universität Hamburg, Fachbereich Informatik, 1994.

Neufeld 93: A. Neufeld. „Validierung konzeptueller Schemata". Dissertation, Universität Karlsruhe, Fakultät für Informatik, 1993.

Nietsch 96: O. Nietsch. „Die Anbindung des Tycoon-Systems an die CASE-Umgebung StP über ein Repository". Studienarbeit, Universität Hamburg, Fachbereich Informatik, 1996.

Oberweis 96: A. Oberweis. *Modellierung und Ausführung von Workflows mit Petri-Netzen.* Teubner, Stuttgart, 1996.

Page 91: B. Page. *Diskrete Simulation. Eine Einführung mit Modula-2.* Springer, Berlin u.a., 1991.

Pnueli, Shalev 91: A. Pnueli und M. Shalev. „What is in a Step: On the Semantics of Statecharts“. In: *T. Ito und A.R. Meyer (Hrsg.). Lecture Notes in Computer Science 780, Springer, Berlin u.a.*, Seite 244–264, 1991.

Raasch 91: J. Raasch. *Systementwicklung mit Strukturierten Methoden.* Hanser, München u.a., 1991.

Reisig 82: W. Reisig. *Petrinetze: Eine Einführung.* Springer, Berlin u.a., 1982.

Rumbaugh et al. 91: J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, und W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

Schröder 97: G. Schröder. „Eine Entwicklungsumgebung für kooperierende persistente Objektsysteme (Arbeitstitel)“. Dissertation, Universität Hamburg, Fachbereich Informatik, 1997.

Sommerville 96: Ian Sommerville. *Software Engineering.* Addison-Wesley, Reading, Massachusetts, 1996.

Stamer 97: S. Stamer. „Entwicklung einer grafischen Workflow-Notation auf Basis der Infrastruktur des StP/OMT-Tools (Arbeitstitel)“. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, 1997.

Thomas, Nejmeh 92: I. Thomas und B.A. Nejmeh. „Definitions of Tool Integration for Environments“. In: *IEEE Software, Nr. 8, New York*, Seite 29–35, 1992.

v.d. Beeck 94: M. v.d. Beeck. „A Comparison of Statecharts Variants“. In: *H. Langmaack, W.P. de Roever und J. Vytopil (Hrsg.). Lecture Notes in Computer Science 863, Springer, Berlin u.a.*, Seite 128–148, 1994.

Versteegen 96: G. Versteegen. „Gute Pflege: Qualitätsmanagement durchgängig realisieren“. In: *iX, Nr. 11, Hannover*, Seite 144–150, 1996.

Wassermann 89: A.I. Wassermann. „Tool Integration in Software Engineering Environments“. In: *F. Long (Hrsg.). Lecture Notes in Computer Science 467, Springer, Berlin u.a.*, Seite 137–149, 1989.