Logische Modellierung von Benutzungsschnittstellen

Realisierung einer Sprache zur Darstellung von Kontexten und Aktionen

Stefan Hügel

Diplomarbeit

Albert–Ludwigs–Universität Freiburg
Institut für Informatik und Gesellschaft
Abteilung 1: Modellbildung und soziale Folgen

in Zusammenarbeit mit

Universität Fridericiana zu Karlsruhe
Fakultät für Informatik
Institut für Logik, Komplexität und Deduktionssysteme

Betreuung:	Prof. Dr. Britta Schinzel, Freiburg Prof. Dr. Peter H. Schmitt, Karlsruhe Dipl.–Inform. Friedrich Strauß, Freiburg
Beginn:	15. Dezember 1993
Ende:	14. Juni 1994

Ich versichere, daß ich diese Arbeit selbständig und ohne die Inanspruchnahme unzulässiger Hilfe angefertigt habe. Die verwendete Literatur ist vollständig im Literaturverzeichnis aufgeführt.

Freiburg i. Brsg., den 14. Juni 1994

Stefan Hügel

Vorbemerkung

Die vorliegende Arbeit ist meine Diplomarbeit im Fach Informatik an der Universität Karlsruhe. Sie wurde durchgeführt am Institut für Informatik und Gesellschaft (IIG) der Universität Freiburg; dabei bestand eine Kooperation mit dem Institut für Logik, Komplexität und Deduktionssysteme an der Karlsruher Fakultät für Informatik.

Für die Aufnahme in ihrer Abteilung und die Betreuung dieser Arbeit danke ich Frau Prof. Dr. Britta Schinzel. Herrn Prof. Dr. Peter H. Schmitt danke ich für die Betreuung als Vertreter der Fakultät für Informatik in Karlsruhe.

Ebenfalls für die Betreuung der Arbeit und dafür, daß er jederzeit für Fragen ansprechbar war, danke ich Frieder Strauß. Ihm und Kirsten Winter danke ich für die gute Zusammenarbeit innerhalb des Projekts SUSI. Herbert Damker danke ich für Hinweise auf Unklarheiten. Allen Mitarbeiterinnen und Mitarbeitern des Instituts für Informatik und Gesellschaft, insbesondere der Abteilung 1, danke ich für die anregende Arbeitsatmosphäre.

Meinen Eltern danke ich dafür, daß sie mir durch vielfältige Unterstützung das Studium — und damit diese Arbeit — ermöglicht haben.

Freiburg, im Juni 1994 *S.H.*

Abstract

In dieser Arbeit wird der Entwurf und die Realisierung einer produktionsbasierten Dialogkomponente zur Steuerung von Benutzungsschnittstellen beschrieben, die mit Konzepten der Aktionstheorie und der Situationstheorie arbeitet.

Dabei wurde insbesondere eine auf diesen Konzepten basierende Beschreibungssprache ausgearbeitet und ein Interpretierer für die Sprache implementiert.

Die Steuerung der Benutzungsoberfläche erfolgt durch die Interpretation von Aktionsregeln, deren Ausführung durch Ereignisse angestoßen wird. Es werden Vorbedingungen abgeleitet und — im Erfolgsfall — eine Wissensbasis gemäß der Nachbedingung der Regel modifiziert. In dieser Wissensbasis ist der Zustand der Oberfläche in Form von prädikatenlogischen Fakten repräsentiert. Abgeleitet wird mit Hilfe der Wissensbasis und zusätzlicher Nebenbedingungen, durch die komplexere Zusammenhänge formuliert werden können.

Die Interaktion der Dialogsteuerungskomponente mit der Anwendung und der Darstellungsschicht erfolgt — außer durch Ereignisse — durch definierte Seiteneffekte: dem Aufruf speziell dafür bereitgestellter Schnittstellenfunktionen.

Das Wissen ist bei dem Ansatz durch Situationen strukturiert, die wiederum unterschiedlichen Situationstypen angehören können. Durch gleiche Situationstypen werden vergleichbare Konzepte formalisiert; sie weisen das gleiche — durch Aktionsregeln formulierte — dynamische Verhalten und die gleichen — durch Nebenbedingungen formulierten — komplexen Zusammenhänge auf. Die Situationen, die zur Laufzeit dynamisch erzeugt werden können, entsprechen den Ausprägungen dieser Konzepte; ihr aktueller Zustand wird durch die jeweils in der Wissensbasis enthaltene Faktenmenge dargestellt.

Es wird eine Implementierung vorgestellt, die unter CLOS (Common Lisp Object System) auf Arbeitsplatzrechnern von Hewlett–Packard realisiert ist.

Inhaltsverzeichnis

1

Einleitung

Im Rahmen eines Projekts am Institut für Informatik und Gesellschaft (Abteilung *Modellbildung und soziale Folgen*) der Universität Freiburg wird ein ereignisbasiertes System zur Realisierung der Dialogkomponente innerhalb der Mensch–Maschine–Schnittstelle entwickelt, das zur Dialogsteuerung auf Konzepte der Situationstheorie und der Aktionstheorie zurückgreift.

Zur Steuerung der Benutzungsoberfläche sollen in diesem System Aktionen im Sinne der Aktionstheorie formalisiert werden können; die Voraussetzungen für deren Ausführung wird dabei aus einer Wissensbasis deduktiv abgeleitet.

Zu diesem Zweck wird ein Formalismus entwickelt, durch den sowohl Anwendungswissen als auch für die Benutzungsoberfläche relevantes Wissen — einschließlich komplexer Zusammenhänge — repräsentiert werden kann. Insbesondere soll es ermöglicht werden, der Benutzerin bzw. dem Benutzer durch die Ausnutzung von Wissen über den momentanen Zustand und das (mögliche) dynamische Verhalten der Oberfläche adäquate, *kontextbezogene* Hilfe zur Verfügung zu stellen.

1.1 Motivation

Nachdem die Bedienung von Datenverarbeitungsanlagen lange Zeit über textuelle Schnittstellen und feste Menümasken erfolgt ist, gewinnen die vor etwa zehn Jahren eingeführten *graphischen Benutzungsoberflächen* immer mehr an Bedeutung. Solche Oberflächen versuchen meist, der Intuition von Benutzerinnen und Benutzern entgegenzukommen, indem sie (logische) Elemente bzw. Objekte in der Rechenanlage (wie z.B. Dateien oder Verzeichnisse des Dateisystems oder auch Peripheriegeräte) durch Symbole auf dem Bildschirm darstellen. Diese Symbole können — i.d.R. durch Mausbewegungen — manipuliert werden; die Bewegungen lösen dann entsprechende Aktionen im darunterliegenden System aus (*direktmanipulative* Schnittstellen).

Neben der Vereinfachung der Benutzung sind diese Systeme auch in der Lage, Arbeitsabläufe flexibler zu gestalten. Erforderten die früher vorherrschenden Bildschirmmasken oft die Einhaltung einer festen Reihenfolge bei der Eingabe von Daten, so kann diese bei direktmanipulativen Schnittstellen weitgehend — abgesehen von etwaigen Erfordernissen der darunterliegenden Anwendung — frei gewählt werden. Damit wird vermieden, daß die Anwendung von Software durch Eigenheiten der Benutzungsoberfläche in unnötiger Weise zusätzlich erschwert wird.

In diesem Zusammenhang spielt die *Software–Ergonomie* eine immer größere Rolle (eine Übersicht zu diesem Thema findet sich in [Balzert et al. 1988] oder in [Maaß 1993]). Für die ergonomische Bewertung existieren mittlerweile Normen (DIN 66234 Teil 8 bzw. ISO 9241 Part 10). In [Oppermann et al. 1992] sind die wesentlichen Anforderungen zusammengestellt.

Wesentliche Aspekte der Software–Ergonomie sind *Transparenz* und *Selbstbeschreibungsfähigkeit* der Systeme. Durch sie wird die Erlernbarkeit der Software maßgeblich gefördert, indem sich die Benutzerin bzw. der Benutzer Funktions– und Anwendungsbereiche erschließen oder die Benutzung vereinfachen kann ([Oppermann et al. 1992]).

Der Begriff der Transparenz wird von Spinas folgendermaßen erklärt (hier zitiert nach [Oppermann et al. 1992]):

> Die *Transparenz* eines System soll dem Benutzer die Bildung eines Struktur– und Prozeßmodells des Systems im Gedächtnis erleichtern, was ihm die notwendigen Orientierungsgrundlagen für die Benutzung bietet. (...) Ein System sollte dementsprechend seine Nutzungs– und Kontrollmöglichkeiten sowie aktuelle Bearbeitungszustände dem Benutzer an der Benutzungsoberfläche strukturiert offenlegen (...)

Den Begriff der Selbstbeschreibungsfähigkeit definiert DIN 66234 so (ebenfalls zitiert nach [Oppermann et al. 1992]; Hervorhebung durch den Autor):

> Ein Dialog ist selbstbeschreibungsfähig, wenn dem Benutzer auf Verlangen Einsatzzweck sowie Leistungsumfang des Dialogsystems erläutert werden können und wenn *jeder einzelne Dialogschritt unmittelbar verständlich ist oder der Benutzer auf Verlangen dem jeweiligen Dialogschritt entsprechende Erläuterungen erhalten kann.*
>
> In Ergänzung zur Benutzerschulung sollen diese Erläuterungen dazu beitragen, daß sich der Benutzer für das Verständnis und für die Erledigung der Arbeitsaufgabe zweckmäßige Vorstellungen von den Systemzusammenhängen machen kann; z.B. über Umfang, Aufgaben, Aufbau und Steuerbarkeit des Dialogsystems, über Benutzung dieser Erläuterungen, über Umgang mit Fehlermeldungen. (...)

Die meisten der heutigen Dialogsysteme bieten zur Erreichung dieser Ziele Hilfe in unterschiedlicher Form an. So existieren z.B. im Betriebssystem UNIX Hilfsseiten (*manual pages*), die die Verwendung und den Zweck von Kommandos beschreiben.

Bei graphischen Oberflächen gibt es oftmals noch weitere Möglichkeiten. So wird zusätzlich zu den Kommandobeschreibungen Hilfe zu jedem einzelnen Bildschirmobjekt angeboten, die — in einem entsprechenden Modus — nach Anklicken des Objekts mit der Maus zur Verfügung steht. Abbildung 1.1 zeigt ein Beispiel aus der HP–VUE–Arbeitsumgebung von Hewlett–Packard.

In der Regel bieten all diese Systeme jedoch nur *statische* Hilfe; *dynamisches* Wissen, wie z.B. Informationen über die im Moment der Hilfeanforderungen möglichen Aktionen, die vom aktuellen Zustand des Systems abhängen, können sie nicht bereitstellen.

Der Grund dafür liegt in prinzipiellen Eigenschaften der meist eingesetzten *ereignisbasierten Systeme*, die Ereignisse an der Benutzungsschnittstelle aufnehmen und — oft mit Hilfe einer Tabelle — damit assoziierte Befehle an die Anwendungsschnittstelle absetzen (*„callbacks“*). Das in der Dialogsteuerung vorhandene Wissen bezieht sich dabei nur auf den unmittelbaren Zusammenhang *Ereignis — Aktion*; darüberhinausgehende Informationen sind im Code der Anwendung verborgen.

Angestrebt wird nun, Wissen zur Anwendung und zur Benutzungsoberfläche zu *integrieren* und darüberhinaus *dynamische Abläufe* in geeigneter Weise zu beschreiben, so daß Informationen

Abbildung 1.1: *Hilfe zu einem Symbol der Benutzungsoberfläche*

darüber explizit vorliegen. Diese Informationen können dann als Grundlage kontextbezogener Hilfsfunktionen dienen. Darüberhinaus soll der dazugehörige Formalismus eine angemessene Beschreibungsmöglichkeit für die Zusammenhänge zwischen Aktionen an der Oberfläche und in der Anwendung bieten.

1.2 Ausgangspunkt

Die vorliegende Arbeit baut auf den Ergebnissen von Strauß ([Strauß 1992], [Strauß 1993a], [Strauß 1993b]) auf, der ein Modell zur situationsorientierten Beschreibung von Benutzungsoberflächen (SUSI — **S**ituation Oriented **Us**er **I**nterface Model) konzipiert hat.

Das Ziel dieser Arbeiten ist es, Kontexte, die bei graphischen Benutzungsschnittstellen z.B. durch Fenster gegeben sind, angemessen darzustellen und damit Benutzerinnen und Benutzern entsprechende Informationen zur Verfügung zu stellen. Dies wird von den meisten ereignisbasierten Systemen nicht geleistet, da hier oft Information über Oberflächen– und Anwendungszustand nur innerhalb der jeweiligen Komponente vorliegt; für adäquate Hilfestellungen notwendiges Wissen ist zum Teil im Anwendungscode verborgen, zum Teil kann es aufgrund der Trennung zwischen den einzelnen Komponenten nicht in geeigneter Weise zusammengeführt werden. In SUSI soll nun dieser Mangel behoben werden, indem Wissen aus der Präsentationsschicht *und* aus der Anwendung gemeinsam repräsentiert wird.

Darüberhinaus wird es ermöglicht, bei der Programmierung der Dialogkomponente die Zusammenhänge im Bereich der Benutzerinteraktion problemorientiert in Form logischer Aussagen (entsprechend dem Paradigma der logischen Programmierung, siehe dazu [Schmitt 1991]) darzustellen.

Das dynamische Verhalten der Oberfläche wird in dem Modell durch *Aktionen* (repräsentiert durch Regeln) definiert, deren Ausführung von einem *Kontext* (repräsentiert durch Nebenbedingungen und elementare Fakten) abhängig ist. Strukturiert wird die Beschreibung durch die Einführung von *Situationen*, die wiederum einzelnen *Situationstypen* zugeordnet sind. Durch Situationen können beispielsweise Bildschirmbereiche (Fenster) dargestellt werden; gleichartige Fenster (z.B. Verzeichnisfenster eines Dateimanagers) gehören dann demselben Situationstyp an.

Die einzelnen Fenster werden also durch Situationen beschrieben, die — gemäß ihrer konzeptuellen Eigenschaften — zu verschiedenen Typen gehören können. Ein Beispiel findet sich in der Abbildung 1.2.

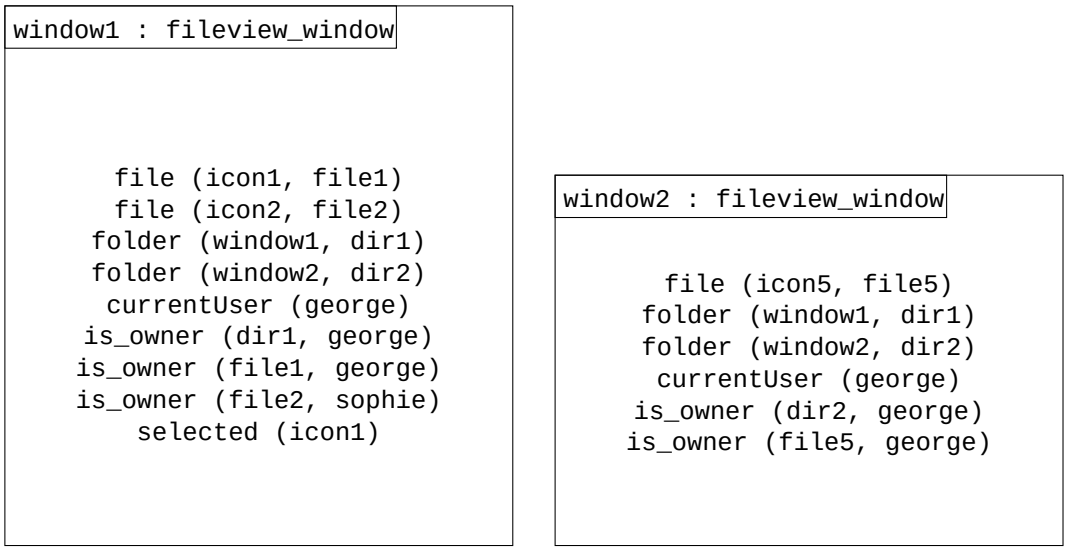

Abbildung 1.2: *Situationen werden durch die in ihnen enthaltenen Fakten festgelegt. Diese Abbildung zeigt einige Fakten von Situationen des Typs* `fileview_window`*, die Fenster eines Dateimanagers repräsentieren. Den Situationen gleichen Typs sind die hier nicht dargestellten Aktionsregeln und Nebenbedingungen gemeinsam.*

Die Situationstypen definieren also die abstrakten Konzepte und das dynamische Verhalten durch Nebenbedingungen (*constraints*) und Aktionsregeln; die Situationen repräsentieren einzelne Ausprägungen des jeweiligen Typs; deren Zustand wird durch eine (in der Zeit veränderliche) Menge von Fakten repräsentiert.

Die wesentlichen Eigenschaften des Ansatzes sind:

- *Die Verbindung von Information über die Schnittstelle und die Anwendung*

  Dies wird dadurch erreicht, daß neben dem dynamischen Verhalten (Aktionen) der Benutzungsoberfläche auch Bedingungen für Anwendungsaktionen festgelegt werden können, beispielsweise in Form einer Nebenbedingung.

- *Explizite Darstellung von Kontexten und Aktionen*

  Das dynamische Verhalten und die dafür notwendigen Bedingungen sind in der Dialogspezifikation vollständig beschrieben; die Zustände, die sich mit der Zeit ändern können, werden durch Faktenmengen explizit dargestellt.

- *Möglichkeit der Bereitstellung kontextbezogener Hilfsfunktionen*

  Ein Modul, das Hilfsfunktionen bereitstellt, kann auf das in der Dialogbeschreibung vorhandene Wissen zugreifen und damit Hilfstexte generieren.

Die angestrebte Hilfe soll zum einen den aktuellen Kontext beschreiben, z.B. den momentanen Verzeichnispfad und Eigenschaften der in diesem Verzeichnis enthaltenen Dateien. Darüberhinaus soll er zum anderen mögliche Aktionen anzeigen, bzw. bei Aktionen, die gerade nicht ausführbar sind, einen Grund dafür liefern. Ein Beispiel für Informationen, die dazu bereitgestellt werden könnten, zeigt Abbildung 1.3 (nach [Strauß 1993b]). Nicht verfolgt wird das Ziel, Pläne zum Erreichen gegebener Ziele zu generieren. Wir folgen damit dem Begriff der situierten Aktion (*situated Action*, nach [Suchman 1987]), der Handeln als auf die momentane Situation bezogen und nicht bis ins Detail geplant auffaßt.

Der aktuelle Kontext ist:	**Mögliche Aktionen sind:**
— Das aktuelle Arbeitsverzeichnis ist `/users/stefan` — Die folgenden Dateien sind ausgewählt: `diplom.dvi` `diplom.ps` — Das aktuelle Verzeichnis gehört `stefan`	— Ziehen des Sinnbilds an eine andere Stelle um eine Kopie der Datei zu erhalten — Ziehen des Sinnbilds an eine andere Stelle um die Datei zu verschieben ist nicht möglich, denn die Datei ist nicht *verschiebbar* — Auswählen von Sinnbildern, um mehrere Sinnbilder an eine andere Stelle ziehen — Doppeltes Anklicken des Sinnbilds um die Datei zum Edieren zu öffnen — Aktivieren einer Menüfunktion für die Datei: *edieren*, *ansehen*, *drucken*

Abbildung 1.3: *Für die Bereitstellung kontextabhängiger Hilfe interessierende Informationen*

1.3 Ziele dieser Arbeit

Die Grundkonzepte der situationsorientierten Modellierung von Benutzungsschnittstellen wurden im vorigen Abschnitt dargestellt. Außerdem wurde das Projekt Susi kurz beschrieben. Die Ziele der vorliegenden Arbeit sind nun:

- Die detaillierte Ausarbeitung der vorgestellten Beschreibungssprache sowie die Behandlung zugrundeliegender theoretischer Konzepte: der Aktions– und der Situationstheorie.
- Der Entwurf und die Implementierung eines Interpretierers, der nach Maßgabe einer Dialogspezifikation in dieser Sprache die Benutzungsoberfläche steuert. Dies umfaßt einerseits die Umsetzung von Aktionen an der Benutzungsschnittstelle in Anwendungsbefehle, andererseits die Darstellung von Rückmeldungen an der Oberfläche. Zusätzlich werden Berechnungen ausgeführt, d.h. Bedingungen für die Aktionen abgeleitet.
- Zur Illustration und als erste Testanwendung wird ein Dateimanager mit Hilfe von Susi realisiert. Dieser soll auch für eine erste Bewertung herangezogen werden.
- Darüberhinaus wird der Ansatz von Susi innerhalb der bereits existierenden, verwandten Ansätze eingeordnet.

Das auf dem Interpretierer aufsetzende Hilfsmodul sowie die Elemente der Präsentationsschicht (der konkreten Darstellung an der Oberfläche) werden in der parallel laufenden Arbeit [Winter 1994] realisiert.

Das entstehende System soll später dazu dienen, die Brauchbarkeit des situationsbezogenen Ansatzes zu evaluieren. Es soll dabei insbesondere untersucht werden, wo die Einsatzmöglichkeiten und die Grenzen liegen und inwiefern eine verbesserte Selbstbeschreibung des Systems positive Effekte auf die Schulung der Benutzerinnen und Benutzer haben kann.

1.4 Gliederung der Arbeit

Im *ersten Kapitel* wurde eine Motivation für die vorliegende Arbeit gegeben, die bereits geleisteten Vorarbeiten beschrieben sowie Ziele und Aufgabenstellung der Arbeit dargestellt.

Im *zweiten Kapitel* wird in die grundlegenen Konzepte eingeführt, die beim Entwurf von SUSI eingeflossen sind. Zunächst wird ein Überblick über die relevanten Aspekte des Gebiets der graphischen Benutzungsoberflächen gegeben und dann die theoretischen Aspekte — Situations– und Aktionstheorie — beleuchtet.

Im *dritten Kapitel* werden verwandte Ansätze dargestellt: bereits existierende Systeme, die mit einem ähnlichen Aktionskonzept arbeiten.

Im *vierten Kapitel* wird das Konzept vorgestellt, das der Realisierung von SUSI zugrundeliegt. Insbesondere werden dabei Semantik und Syntax erläutert, ein Architekturansatz vorgestellt und ein Vergleich zu den zuvor beschriebenen Ansätzen gezogen.

Im *fünften Kapitel* wird das System beschrieben, das im Rahmen der Arbeit als Prototyp implementiert wurde.

Das *sechste Kapitel* bietet einen Rückblick auf die Arbeit und einen Ausblick auf mögliche weitere Arbeiten im Zusammenhang mit SUSI.

Im *Anhang* finden sich zuletzt die Grammatik der Beschreibungssprache und Beispiele für die Formalisierung des Dialogs.

2

Grundlagen

Wie in der Einleitung bereits dargestellt, soll in dem Projekt SUSI ein Werkzeug bereitgestellt werden, um graphische Benutzungsoberflächen, insbesondere die Dialogsteuerungskomponente, mit Hilfe von Konzepten aus dem Bereich der Situations– und der Aktionstheorie zu konstruieren. Ziel dieses Kapitels ist es, die Grundlagen dieser beiden Themenbereiche zunächst unabhängig voneinander darzustellen. Es gliedert sich demgemäß in zwei Abschnitte:

- Zunächst wird auf den Aufbau von graphischen Benutzungsoberflächen (*Graphical User Interfaces*, GUI) und Managementsystemen für Benutzungsschnittstellen (*User Interface Management Systems*, UIMS) eingegangen. Dabei werden Anforderungen an solche Systeme skizziert, die Architektur vorgestellt und prinzipielle Ansätze der Dialogsteuerung beschrieben.
- Inhalt des zweiten Abschnitts sind die theoretischen Ausgangspunkte der Arbeit, die Situationstheorie und die Aktionstheorie. Deren Grundzüge werden dargestellt und die Beziehungen der SUSI zugrundeliegenden Konzepte zu diesen Theorieansätzen diskutiert.

Die Ergebnisse des Kapitels bilden die Grundlage für Konzeption und Realisierung des im Rahmen dieser Arbeit entwickelten Systems.

2.1 Graphische Benutzungsschnittstellen

Dieser Abschnitt beschäftigt sich mit Modellen und Werkzeugen zur Erstellung graphischer Benutzungsschnittstellen. Dabei werden zunächst Benutzungsschnittstellen–Entwicklungssysteme (*User Interface Development Systems*, UIDS) betrachtet. Dies sind Systeme, die verschiedene Arten der Unterstützung bei der Erzeugung von Benutzungsoberflächen anbieten.[1] Die Steuerung der Schnittstelle zur Laufzeit übernehmen die Benutzungsschnittstellen–Management–Systeme. Auf solche Systeme wird anschließend eingegangen.

[1] Myers ([Myers 1989], [Myers 1993]) differenziert zwischen UIDS, worunter er alle Werkzeuge und Systeme zur Schnittstellenkonstruktion faßt, und UIMS, dem für die Steuerung zur Laufzeit zuständigen Systemteil. Diese Differenzierung wird jedoch nicht allgemein in der Literatur verwendet; teilweise wird ohne weitere Unterscheidung von UIMS gesprochen.

Die Unterstützung der Oberflächenentwicklung durch UIDS deckt idealerweise das gesamte Spektrum der Entwicklung ab, das von der Anforderungsanalyse über Entwurf und Implementierung bis zur Bewertung reicht. Davon ausgehend sind nach Myers ([Myers 1989], [Myers 1993]) an ihre Funktionalität die folgenden Anforderungen zu stellen:

- Unterstützung von Eingabegeräten wie beispielsweise einer Maus
- Auswertung der Eingaben von Benutzerinnen und Benutzern
- Unterstützung des Sequentialisierens von Operationen
- Geeignetes Reagieren auf fehlerhafte Eingaben
- Bearbeiten von Anforderungen, die Ausführung von Operationen abzubrechen (*abort*) oder Eingaben rückgängig zu machen (*undo*)
- Erzeugung geeigneter Rückmeldungen als Bestätigung für Eingaben
- Bereitstellung geeigneter Hilfe bei der Benutzung
- Erzeugung einer entsprechenden Rückmeldung an der Oberfläche bei Änderungen in den Daten durch die Anwendung
- Darstellung von Änderungen wenn Daten der Anwendung durch die Benutzerin bzw. den Benutzer aktualisiert werden
- Unterstützung von graphischer Gestaltung und Layout der Oberfläche
- Unterstützung von Ausschnittsverschiebungen (z.B. durch Verschiebebalken) und Edieren
- Befreiung der Anwendung von der Steuerung der Benutzungsoberfläche, d.h. alle derartigen Funktionen müssen vom UIDS zur Verfügung gestellt werden
- Unterstützung einer Bewertung (Evaluation) der Benutzungsschnittstelle, z.B. im Hinblick auf einfache Benutzbarkeit und Erlernbarkeit
- Ermöglichung von Anpassungen der Oberfläche an den persönlichen Bedarf (Individualisierbarkeit)

Um diese Ziele zu erreichen, stellen UIDS unterschiedliche Werkzeuge zur Verfügung. Dabei ist es je nach System unterschiedlich, was tatsächlich zur Verfügung steht. Ein UIDS kann zur Erfüllung der genannten Aufgaben folgende Komponenten umfassen:

- Einen Werkzeugkasten (*toolkit*), der inbesondere die Erzeugung von graphischen Elementen wie Fenster mit Verschiebebalken, Dateisymbolen u.ä. unterstützt,
- Eine Komponente zur Dialogüberwachung, die Ereignisse verarbeitet und Interaktionen steuert,
- Eine Programmierumgebung, die Unterstützung bei der Strukturierung der Schnittstellenbeschreibung und der Verbindung zur Anwendung bietet,
- Einen (mausbasierten) Layout–Editor, mit dem die Positionierung der graphischen Oberflächenelemente spezifiziert werden kann,
- Eine Komponente zur Analyse, die entweder eine automatische Bewertung vornimmt oder Informationen zur späteren Auswertung sammelt.

2.1.1 Architektur von graphischen Benutzungsschnittstellen

Lange Zeit wurden Benutzungsoberflächen zusammen mit der Anwendung realisiert; dabei wurde der Code der Oberfläche mit dem Code der Anwendung verwoben. Dies führte jedoch oft zu schlecht wartbaren und uneinheitlichen Oberflächen; Programmodule wurden von verschiedenen Programmiererinnen und Programmierern implementiert, die unterschiedliche Vorstellungen von guten Benutzungsschnittstellen hatten (ein Beispiel für das Ergebnis findet sich in [Hartson, Hix 1989], wo beispielsweise die Anforderung von Hilfsseiten in jedem Programmodul durch eine andere Tastensequenz erfolgt). Deswegen gab es später Bestrebungen, die Oberflächenkomponente von der Anwendung zu trennen. Dies führte zum *Seeheim–Modell*, bei dem die drei Systemteile *Präsentationskomponente*, *Dialogsteuerung* und *Anwendungsschnittstelle* weitgehend voneinander unabhängig sind (Abbildung 2.1, nach [Myers 1989], [Olsen 1992]).

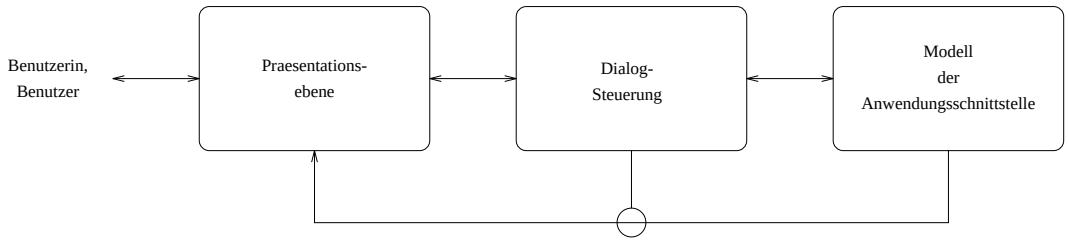

Abbildung 2.1: *Seeheim–Architekturmodell für Benutzungsschnittstellen*

Die einzelnen Komponenten des Systems erfüllen in diesem Modell die folgenden Funktionen:

- Die Präsentationskomponente besteht aus allen Teilen, die für Benutzerinnen und Benutzer direkt an der Oberfläche sichtbar sind. Dies umfaßt das optische Erscheinungsbild sowie die Steuerung der physischen und logischen Eingabeeinheiten (der *„look“* der Schnittstelle).
- Die Dialogsteuerung ist das Herzstück des Systems. Sie interpretiert die Eingaben und gibt die entsprechenden Anforderungen an die Anwendung weiter. Hier wird festgelegt, in welcher Weise dies geschieht, d.h. auf welche Weise Eingaben an der Oberfläche mit Aktionen der Anwendung verknüpft sind (das *„feel“* der Schnittstelle).
- Die Anwendungsschnittstelle legt fest, in welcher Weise von der Benutzungsschnittstelle auf die eigentliche Anwendung zugegriffen werden kann.

Die im Seeheim–Modell vorgesehene strikte Trennung erweist sich jedoch als problematisch, insbesondere im Hinblick auf *semantische Rückmeldungen*, die für eine *direkte Manipulation* ([Shneiderman 1987], [Ilg, Ziegler 1988]) von Oberflächenobjekten notwendig sind. Soll etwa in einem Fenstersystem eine Datei kopiert werden, wird sie mit der Maus auf das Symbol des Zielordners bewegt. Um anzuzeigen, daß hier eine Aktion (das Kopieren) möglich ist, wird das Symbol invertiert dargestellt, was bei einem Dateisymbol in dieser Situation nicht geschehen würde. Die Information, welche Symbole Ordner, und welche Symbole Dateien repräsentieren, ist i.d.R. in der Anwendung vorhanden; die Präsentationsebene muß jedoch auf diese Information zugreifen können.

Aus diesen Gründen ist man vom Seeheim–Modell in seiner ursprünglichen Form wieder abgerückt. Ein verändertes Modell für die Architektur zeigt Abbildung 2.2 ([Myers 1989]). Hier

werden Präsentationskomponente und Dialogsteuerung wieder enger aneinander gekoppelt und eine Komponente zur semantischen Unterstützung eingefügt. So wird die notwendige Kommunikation zwischen Oberfläche und Anwendung ermöglicht.

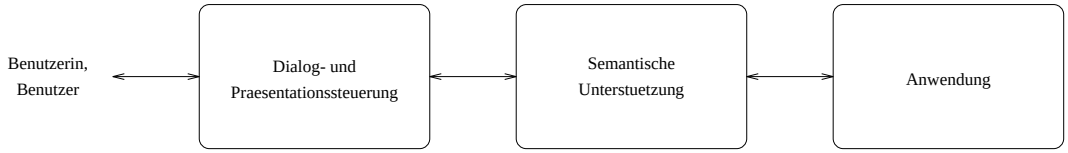

Abbildung 2.2: *Modifiziertes Modell zur Unterstützung semantischer Rückmeldungen*

Wenn auch das Seeheim–Modell sich als Modell für die tatsächliche Architektur der Software als ungeeignet erwiesen hat, so lassen sich die bei der Realisierung von Benutzungsschnittstellen auftretenden Fragestellungen damit weiterhin gut strukturieren.

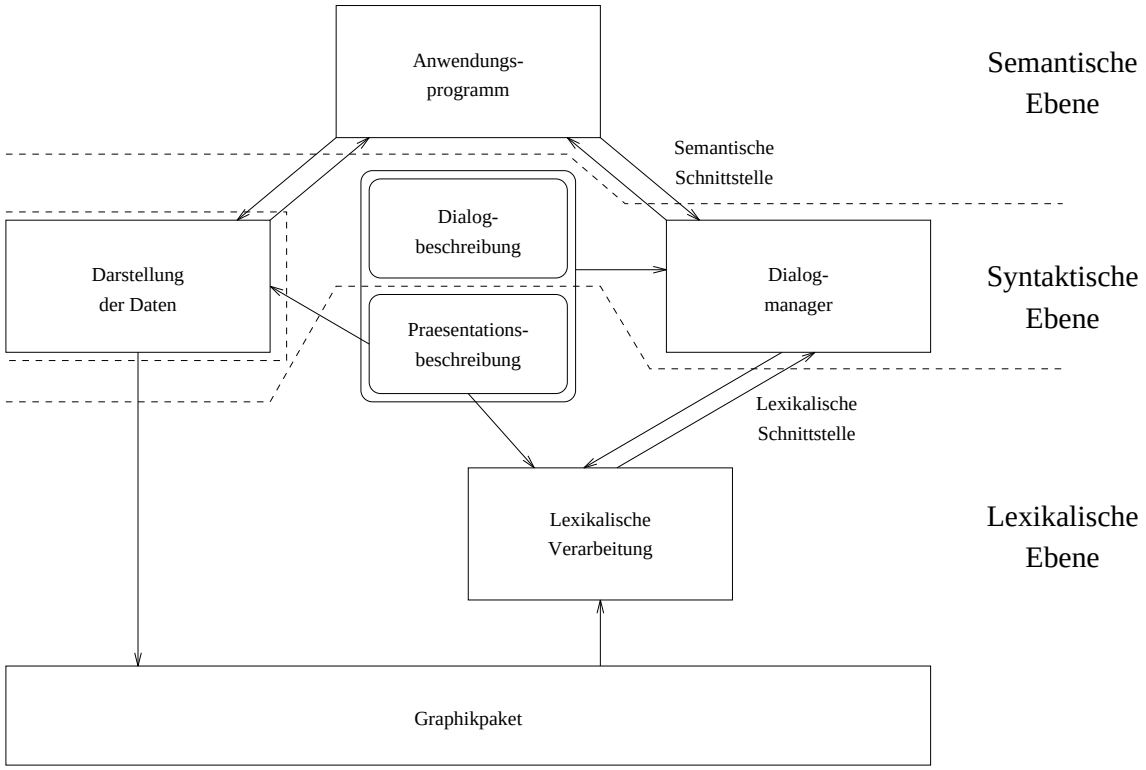

Abbildung 2.3: *Übersicht über die Struktur von Managementsytemen für Benutzungsoberflächen*

Die in Abbildung 2.3 ([Olsen 1992]) gezeigte Architektur spiegelt die einzelnen Strukturelemente bei der Entwicklung graphischer Benutzungsschnittstellen wider. Der lexikalischen Ebene ordnet man dabei die Elemente zu, die unmittelbar an der sichtbaren Oberfläche bzw. der Benutzerinteraktion beteiligt sind, also die *Elemente der graphischen Darstellung* und die *lexikalische Verarbeitung*. Deren Spezifikation erfolgt in der *Präsentationsbeschreibung*. Die syntaktische Ebene umfaßt den *Dialogmanager*, dessen Operationen durch die *Dialogbeschreibung* definiert werden. Die semantische Ebene besteht hauptsächlich aus der *Anwendung* selbst und deren *Schnittstelle* zum Dialogmanager. Hier geschieht die Umsetzung in konkrete Anwendungsopera-

tionen (semantische Aktionen). Die in der Abbildung gezeigte Komponente zur Darstellung der Daten ist in diesem Modell ursprünglich nicht vorgesehen; mitunter wird sie der syntaktischen Ebene zugerechnet.

Lexikalische Ebene

Die lexikalische Ebene ist prinzipiell eine Menge *logischer Eingabeeinheiten*, die eine Schnittstelle zu den physischen (Tasten, Drehknöpfe) oder virtuellen (mit der Maus zu bedienenden Bildschirm„knöpfen“, Verschiebebalken) Eingabeeinheiten bieten. Die Eingabeeinheiten werden identifiziert und die Werte, die sie liefern werden gelesen. Bereitgestellt werden Werte wie z.B. die Stellung eines Drehknopfes bzw. Verschiebebalkens sowie ggf. ein Ereignis, das anzeigt, daß der Wert in diesem Moment für die Software relevant ist, d.h. daß aufgrund einer Änderung des Wertes eine entsprechende Reaktion in der Software erfolgen soll.

Die Bindung von logischen an physische (oder virtuelle) Eingabeeinheiten hängt von dem momentanen Kontext ab, z.B. dem gerade aktivierten Fenster. Dieser legt fest, wohin die Eingaben in diesem Moment übermittelt werden. In einem Kontext vorhandene Elemente werden als *acquired* (im Gegensatz zu *released*) bezeichnet. Innerhalb eines Kontexts können darüberhinaus Elemente zwar *acquired* sein, aber trotzdem — aufgrund des Zustands der Anwendung — nicht aktivierbar. (Beispielsweise kann ein Menüpunkt `move` nur gewählt werden, wenn zuvor eine Datei oder ein Verzeichnis selektiert wurde.) Hier wird zwischen *enabled* und *disabled* unterschieden.

Die wesentlichen Aufgaben der lexikalischen Verarbeitung umfassen damit

- die Festlegung, ob eine Eingabeeinheit vorhanden ist oder nicht (*acquired* oder *released*),
- die Festlegung, ob sie aktivierbar ist oder nicht (*enabled* oder *disabled*),
- die Entgegennahme von Eingabeereignissen.

Syntaktische Ebene

Wesentlicher Teil der syntaktischen Ebene ist der Dialogmanager. Er ist für die Steuerung der gesamten Benutzungsschnittstelle verantwortlich. Seine Hauptfunktion ist dabei die Entgegennahme von Eingaben (Ereignissen) von der lexikalischen Verarbeitungseinheit und die Aktivierung semantischer Aktionen einschließlich der Übermittlung der dafür relevanten Daten. Darüberhinaus informiert er die logischen Eingabeeinheiten über ihren Zustand (*acquired/released* bzw. *enabled/disabled*). Spezifiziert wird das Vorgehen des Dialogmanagers in der Dialogbeschreibung.

Die Kommunikation an der lexikalischen wie an der semantischen Schnittstelle erfolgt dabei durch das Übermitteln von Datenstrukturen (*event records*), die jeweils die Art des Ereignisses und die dazugehörige Information enthalten, wie z.B. *„Mausklick auf das Symbol* file“ an der lexikalischen oder *„Kopiere die Datei* file *in das Verzeichnis* folder“ an der semantischen Schnittstelle.

Die Realisierung der syntaktischen Ebene — des Dialogmanagers — wird der Hauptinhalt dieser Arbeit sein.

Semantische Ebene

Die semantische Ebene umfaßt die Schnittstelle zur Anwendung sowie die Anwendung selbst. Für die Anwendungsschnittstelle gibt es zwei prinzipielle Modelle:

- Das *Kommandomodell*, bei dem die Anwendung eine Menge von Kommandoprozeduren bereitstellt
- Das *Datenmodell*, bei dem die Anwendung als Datenstruktur gesehen wird, die durch die Benutzungsschnittstelle manipuliert wird

Semantische Aktionen werden angestoßen, indem die Art der Aktion und zugehörige Daten an die Anwendung weitergegeben werden. Dies kann z.B. in Form geeigneter Datenstrukturen (`record` oder (in C) `struct`) geschehen. In interpretierten Sprachen wie Lisp können auch Funktionsausdrücke als Zeichenketten übergeben werden.

2.1.2 Ansätze zur Realisierung der Dialogsteuerung

Die Dialogsteuerungskomponente in Benutzungsschnittstellen läßt sich auf verschiedene Weisen realisieren. Hier sollen kurz drei Ansätze vorgestellt werden: der zustandsbasierte, der ereignisbasierte und der produktionsbasierte Ansatz.

Zustandsbasierte UIMS

Bei der zustandsbasierten Dialogsteuerung sind mögliche Eingaben vom gerade aktuellen Zustand des Systems abhängig. Eingaben verursachen Zustandsübergänge sowie zusätzlich semantische Aktionen.

Für moderne Benutzungsschnittstellen, die oft sehr viele verschiedene Eingaben zur gleichen Zeit zulassen, ist dieser Ansatz allerdings nicht geeignet. Zustände benötigen für jede mögliche Eingabe eine Ausgangskante. Dadurch kommt man zu sehr umfangreichen und komplexen Übergangsdiagrammen. Allein für die einfache Aufgabe, n Eingaben in beliebiger Reihenfolge zuzulassen, sind bereits 2^n Zustände notwendig, wie in Abbildung 2.4 beispielhaft für $n = 3$ zu sehen ist. Aus diesem und noch anderen Gründen spielen rein zustandsbasierte Ansätze fast keine Rolle mehr.

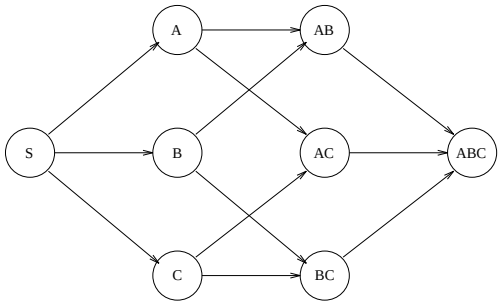

Abbildung 2.4: *Zustandsdiagramm zur Behandlung ungeordneter Eingaben. Für drei Eingaben in beliebiger Reihenfolge sind bereits* $8 = 2^3$ *Zustände notwendig.*

Ereignisbasierte UIMS

Im Gegensatz zu zustandsbasierten Ansätzen (und auch grammatikbasierten), die Eingaben abhängig von der zeitlichen Reihenfolge interpretieren, ist bei neueren Ansätzen — vor allem für Schnittstellen mit direkter Manipulation — die Bildschirmkoordinate entscheidend, an der eine Eingabe (z.B. ein Mausklick) erfolgt. Mögliche Eingaben hängen nicht davon ab, was nach den vorherigen Eingaben *zu diesem Zeitpunkt* möglich ist, sondern von den Bildschirmobjekten, die *an der Oberfläche sichtbar* sind. Dies führt zu ereignisbasierten Systemen.

Aktionen der Benutzerin bzw. des Benutzers erzeugen eine Datenstruktur (*event record*), die an die Dialogsteuerung übermittelt wird. Sie enthält Informationen, wie z.B. die Art des Ereignisses (Mausklick o.ä.) und die dazugehörigen Koordinaten oder das betroffene Objekt. Die so übergebenen Daten werden von der Dialogsteuerung interpretiert und die entsprechenden Aktionen angestoßen.

Die Beschreibungen der Zusammenhänge erfolgen oftmals durch *Ereignistabellen*, in denen Sinnbilder, Ereignisse und die dazugehörigen Funktionen verknüpft sind.

Die zwei Haupteigenschaften ereignisbasierter Ansätze:

- Abhängigkeit von sichtbaren Bildschirmobjekten (anstelle von Zuständen) und
- direkter Zusammenhang zwischen Ereignissen und semantischen Aktionen

machen diesen Ansatz besonders geeignet für graphische Benutzungsschnittstellen. Problematisch ist dieser Ansatz jedoch dann, wenn aufeinanderfolgende Ereignisse voneinander abhängig sind; beispielsweise zunächst die Auswahl eines Befehls aus einem Menü erfolgt und anschließend dieser Befehl auf ein Objekt angewendet werden soll. Der ereignisbasierte Ansatz stellt keine Möglichkeit bereit, die Information, *welcher* Befehl ausgewählt wurde, zu speichern. Dies muß dann in der Anwendung geschehen.

Produktionsbasierte UIMS

Produktionsbasierte UIMS basieren auf *Regeln*, mit deren Hilfe die Benutzungsoberfläche gesteuert wird. Meist wird dieses regelbasierte Konzept in Verbindung mit Ereignissen verwendet.

Regeln sind die Hauptelemente der Dialogsteuerung solcher Systeme. Aufgrund des Eintretens bestimmter Bedingungen lösen sie entsprechende Aktionen aus. Sie haben normalerweise die Form

$$Bedingung \longrightarrow Aktion$$

d.h. es werden zunächst Vorbedingungen geprüft und abhängig davon Aktionen ausgeführt. Die Verbindung zum ereignisbasierten Ansatz schafft dabei eine bestimmte Kategorie von Elementen (Relationen) in der Vorbedingung. Diese stellen Ereignisse dar, die an der Benutzungsoberfläche stattfinden.

Die Vorbedingungen repräsentieren dabei Systemzustände, die zur Ausführung der Regel eingetreten sein müssen. Dies können z.B. Relationen in einer Faktenbasis sein, gesetzte bzw. nicht gesetzte Flags oder Vektoren in einem Zustandsraum.

Das produktionsbasierte Prinzip wird auch in SUSI verwendet. Als Beispiele für weitere Systeme mit diesem Ansatz werden in Kapitel 3 SASSAFRAS ([Hill 1986]), PPS ([Olsen], [Olsen 1992]) und UIDE ([Foley et al. 1989], [Gieskens, Foley 1991]) diskutiert.

2.2 Situationstheorie und Aktionen

Der in dieser Arbeit verwendete situationsorientierte Ansatz ist von den Ergebnissen zweier Forschungsbereiche inspiriert:

- der *Situationstheorie*, die entwickelt wurde, um eine geeignete Methode zur Modellierung von Information und zur formalen Beschreibung von Perzeption zu haben ([Barwise, Perry 1983], [Devlin 1991])
- der *Aktionstheorie* (*Theory of Action*), die sich mit Aktionen beschäftigt, die Transformationen, abhängig von (Vor–) Bedingungen, beschreiben ([Puppe 1988], [Lifschitz 1987], [Pednault 1986], [Pednault 1989], [Shoham 1986], [Kartha 1993]).

Auf den folgenden Seiten sollen nun diese beiden Gebiete kurz dargestellt werden. Anschließend werden die Bezüge zu SUSI herausgearbeitet.

2.2.1 Situationstheorie

Die Situationstheorie[2] wurde als Grundlage für eine formale *Theorie der Information* entwickelt, die vor allem dazu dienen soll, Perzeption zu formalisieren. Mit diesem Ziel wurde ein Rahmen erarbeitet, mit dessen Hilfe Information dargestellt werden kann. Dabei wird davon ausgegangen, daß die einem Menschen verfügbare Information zum einen von dem aktuellen Kontext (der *Situation*) abhängt, und zum anderen sie nicht vollständig explizit vorliegt, sondern auf Hintergrundwissen (*Constraints*) zurückgegriffen wird. Es ist dabei (für ein reales Individuum[3]) nie die gesamte Information verfügbar.

Infons und Fakten

Zur Darstellung von Information wird der Begriff der *Infons* eingeführt. Infons beschreiben die atomaren Einheiten von Information. Das zur Verfügung stehende Wissen ist immer eine (explizit oder implizit vorliegende; siehe dazu später) Menge solcher Infons.

Sie sind dabei den aus der herkömmlichen Prädikatenlogik bekannten Prädikaten (bzw. Relationen) vergleichbar. Ein Infon hat folgendes Aussehen:

$$\ll R, a_1, a_2, \ldots, a_n, i \gg$$

In diesem Beispiel wäre R eine n–stellige Relation, $a_1, a_2, \ldots, a_n$ geeignete Argumente für R und $i \in \{0, 1\}$ die *Polarität* des Infons; eine 1 an dieser Stelle sagt aus, daß $a_1, \ldots, a_n$ in der Relation R stehen; eine 0 sagt aus, daß sie *nicht* in der Relation R stehen. Die Infons $\ll betreut, Frieder, Stefan, 1 \gg$ und $\ll betreut, Frieder, Herbert, 0 \gg$ treffen in diesem Sinne Aussagen über die Betreuung von Diplomarbeiten am IIG.

Infons können dabei sowohl wahr als auch falsch sein; geben sie eine *wahre* Information wieder, so heißen sie *Fakten*.

[2] Die Darstellung hier folgt [Devlin 1991].

[3] Im Gegensatz zur idealisierten Beobachterin/zum idealisierten Beobachter; Devlin verwendet die Begriffe *agent scheme — theorist's scheme*.

Situationen

Das wesentliche Element der Situationstheorie ist der Begriff der *Situation*. Die „Welt“, in der sich ein Individuum bewegt, ist aus einer Menge solcher Situationen zusammengesetzt. Sie sind strukturierte Ausschnitte aus dieser Welt, wie z.B. ein Fußballspiel, das man besucht (und über das man sich später unterhält), oder das Leben einer bestimmten Person (und alles, was damit zusammenhängt, wie Vorfahren, Nachkommen etc.). Situationen werden nicht notwendigerweise bewußt wahrgenommen; Individuen unterscheiden zwischen Situationen, indem sie sich in unterschiedlichen Situationen unterschiedlich verhalten (z.B. im persönlichen Gespräch anders als in einer politischen Debatte).

Aus Situationen kann nun Information (Infons) abgeleitet werden. In Analogie zur Prädikatenlogik ist die in einer Situation (etwa s) enthaltene Information ein *Modell* für bestimmte Infons (z.B. σ), wenn σ in s gültig ist. Dieser Sachverhalt wird durch den Ausdruck

$$s \models \sigma$$

ausgedrückt.

Gleiches gilt für Mengen von Infons: Ist I eine Menge von Infons, so gilt

$$s \models I$$

genau dann, wenn $s \models \sigma$ für alle $\sigma \in I$.

Offensichtlich ist die *Modell*–Relation von der Situation abhängig; die beiden oben gemachten Aussagen über Diplomarbeiten am IIG gelten in der Situation *„IIG im Jahr 1993“*; über die Situation *„IIG im Jahr 1995“* ist nichts ausgesagt.

Eine Situation kann zuletzt auch die Welt als ganzes sein. Dies ist dann die maximale Situation, die jede andere Situation als Teilmenge enthält. Damit läßt sich der Begriff des Fakts formal fassen: steht w für diese maximale Situation, so gilt

$$\phi \; ist \; ein \; Fakt \; genau \; dann, \; wenn \; w \models \phi$$

Abstrakte Situationen

Um nun den Begriff der Situation mathematisch handhabbar zu machen, geht man zu *abstrakten* Situationen über:

> Definition 2.2.1 (Abstrakte Situation) *Eine abstrakte Situation ist eine Menge von Infons.*

Im Gegensatz zur realen Situation, die den für ein gegebenes Individuum sichtbaren Weltausschnitt darstellt, ist eine abstrakte Situation eine mathematische Konstruktion. Das Verhältnis zwischen realer und abstrakter Situation kann idealisiert als

$$s_{abstrakt} = \{\sigma \mid s_{real} \models \sigma\}$$

ausgedrückt werden. Allerdings exstieren abstrakte Situationen, die kein reales Gegenstück haben, wie z.B. Situationen mit widersprüchlichen Aussagen, die in dem Konzept nicht ausgeschlossen werden (siehe dazu unten).

Ein Beispiel für eine abstrakte Situation ist

$$s = \{\ll betreut, Frieder, Stefan, 1 \gg, \ll betreut, Frieder, Herbert, 0 \gg\}$$

(die eine Teilmenge einer großen Zahl umfangreicherer abstrakter Situationen sein kann).

Die *Modell*–Relation ist bei abstrakten Situationen reduziert auf die Elementbeziehung; es gilt also

$$s \models \sigma \ genau\ dann,\ wenn\ \sigma \in s$$

Zwei Definitionen fehlen bei der Beschreibung abstrakter Situationen noch: Abstrakte Situationen waren nicht als widerspruchsfrei vorausgesetzt; d.h. es kann Situationen geben, so daß

$$s \models \{\ll R, a_1, \ldots, R_n, 1 \gg, \ll R, a_1, \ldots, R_n, 0 \gg\}$$

Solche Situationen nennt Devlin *inkohärent*. Um sie auszuschließen, wird der Begriff der *kohärenten Situationen* eingeführt:

> Definition 2.2.2 (Kohärente Situation) *Eine abstrakte Situation ist kohärent, wenn gilt:*
>
> - *Es gibt keine $R, a_1, \ldots, a_n$ so, daß $s \models \ll R, a_1, \ldots, a_n, 1 \gg$ und gleichzeitig $s \models \ll R, a_1, \ldots, a_n, 0 \gg$.*
> - *Gibt es a, b so, daß $s \models \ll same, a, b, 1 \gg$, so gilt $a = b$ (same steht dabei für die Gleichheit).*
> - *Es gibt kein a so, daß $s \models \ll same, a, a, 0 \gg$.*

Reale Situationen sind immer kohärent. Zwei abstrakte Situationen s und s' heißen *kompatibel*, wenn ihre Vereinigung eine kohärente Situation ist.

Dies reicht noch nicht aus, damit eine abstrakte Situation ein reales Gegenstück hat. Um das zu erreichen, werden *tatsächliche* (*actual*) Situationen definiert:

> Definition 2.2.3 (Tatsächliche Situation) *Eine abstrakte Situation heißt tatsächlich, wenn folgendes gilt:*
>
> - *Ist $s \models \ll R, a_1, \ldots, a_n, 1 \gg$, so stehen in der realen Welt tatsächlich $a_1, \ldots, a_n$ in der Relation R.*
> - *Ist $s \models \ll R, a_1, \ldots, a_n, 0 \gg$, so stehen in der realen Welt $a_1, \ldots, a_n$ nicht in der Relation R.*

Offensichtlich ist jede tatsächliche abstrakte Situation kohärent.

Situationstypen

Um nun dieses Modell weiter zu strukturieren, können Situationen aufgrund von Gemeinsamkeiten zusammengefaßt werden. Diese Gemeinsamkeiten sind beliebig; sie werden sich von Individuum zu Individuum unterscheiden. Die dabei entstehenden Kategorien von Situationen werden *Situationstypen* genannt.

Hat man beispielsweise zwei Situationen s_0 und s_1, für die gilt:

$$\begin{aligned} s_0 &\models \ll betreut, Frieder, Stefan, Diplomarbeit, Freiburg, 1 \gg \\ s_1 &\models \ll betreut, Klaus, Martin, Promotion, Hamburg, 1 \gg \end{aligned}$$

so haben diese, bei allen Unterschieden, zumindest eine Gemeinsamkeit: es handelt sich in beiden Fällen um ein Betreuungsverhältnis. Der Begriff des Situationstyps dient dazu, solche Gemeinsamkeiten innerhalb der Theorie zu repräsentieren. Vom gleichen Typ wären in dieser Sicht also alle Situationen, aus denen Infons der Form $\ll betreut, x_1, x_2, x_3, x_4, 1 \gg$ für geeignete x_1, x_2, x_3, x_4 ableitbar sind.

Formal ausgedrückt: Gegeben sei eine Menge I von Infons und ein Parameter $\dot{s}$, der eine Situation bezeichnet. Dann können alle Situationen aus der Menge

$$\{\dot{s} \mid \dot{s} \models I\}$$

einem Typ zugeordnet werden. Dieser Typ wird dann notiert als

$$[\dot{s} \mid \dot{s} \models I]$$

Die Menge I kann dabei beliebig gewählt werden. Die Punkte über den Parametern bezeichnen dabei freie Variablen; siehe dazu [Devlin 1991].

Constraints

Die in einer Situation enthaltenen Infons allein reichen noch nicht aus, um sämtliche vorhandenen Informationen wiederzugeben. Ein Individuum besitzt immer ein bestimmtes Hintergrundwissen, mit dessen Hilfe weitere Aussagen abgeleitet werden können. So wird z.B. eine Person, die aus einem Haus Rauch aufsteigen sieht, daraus ableiten, daß es in diesem Haus brennt. Angewendet wird hier eine Bedingung (*Constraint*) aus dem Hintergrundwissen, nämlich daß Rauch Feuer bedeutet.

Dies wird als Zusammenhang zwischen zwei Situationstypen beschrieben. Der Zusammenhang besteht zwischen den Situationen eines Typs S_0, in dem Rauch aufsteigt und Situationen eines Typs S_1, in dem (zur gleichen Zeit und am gleichen Ort) ein Feuer brennt. Er kann folgendermaßen dargestellt werden:

$$S_0 \Rightarrow S_1$$

wobei

$$S_0 = [\dot{s_0} \mid \dot{s_0} \models \ll Rauch\_vorhanden, \dot{l}, \dot{t}, 1 \gg]$$

und

$$S_1 = [\dot{s_1} \mid \dot{s_1} \models \ll Feuer\_vorhanden, \dot{l}, \dot{t}, 1 \gg]$$

Es wird dann folgendermaßen geschlossen: Wird eine Situation gesehen, in der Rauch aufsteigt (die vom entsprechenden Typ ist), so folgt daraus, daß (gleichzeitig) eine weitere Situation besteht, in der ein Feuer zu beobachten ist.

2.2.2 Aktionstheorie

Die eben umrissene Situationstheorie ist ausreichend, um die in einem gegebenen Kontext vorhandene Information zu beschreiben. Devlin ([Devlin 1991]) weist darauf hin, daß ein Individuum verschiedene Situationen *durch unterschiedliches Verhalten* voneinander abgrenzt. Die Beschreibung des Verhaltens (in bestimmten Situationen) ist jedoch nicht Gegenstand der Situationstheorie.

Die Beschreibung von Aktionen, die in einer Welt bzw. einem Weltausschnitt ausgeführt werden können, ist Gegenstand der Aktionstheorie (*Theory of Action*). Sie beschreibt Änderungen in einem Gegenstandsbereich durch Transformationen auf einer logischen Beschreibung dieses Bereichs, d.h. durch Modifikationen der Interpretation der diesen Bereich beschreibenden Relationen (Prädikate). Die Theorie, die sich dabei herausgebildet hat, soll vor allem bei der Lösung von Planungsproblemen helfen.

Generell besteht ein mit Aktionsregeln arbeitendes System aus den folgenden Komponenten (nach [Puppe 1988]):

- der *Datenbasis*, die die gültigen Fakten enthält
- den *Regeln* zur Herleitung neuer Fakten
- dem *Regelinterpretierer* zur Steuerung des Herleitungsprozesses

Dabei kann eine Herleitung grundsätzlich nach zwei verschiedenen Verfahren erfolgen:

- Bei der *Vorwärtsverkettung* werden um ein Ziel zu erreichen, ausgehend von einer Datenbasis, Regeln ausgeführt, deren Vorbedingung erfüllt ist.
- Bei der *Rückwärtsverkettung* wird vom zu erreichenden Ziel ausgegangen und rückwärts Regeln überprüft, durch die dieses Ziel erreicht werden kann. Damit wird auf die für das Ziel notwendigen Vorbedingungen geschlossen.

Da in unserem Fall nicht untersucht werden soll, wie ein gegebenes Ziel zu erreichen ist, sondern vielmehr ein System durch die Ausführung von Regeln gesteuert wird, ist hier nur die Vorwärtsverkettung von Interesse.

Bei der Interpretation der Regeln wird zunächst die Menge der jeweils anwendbaren Regeln bestimmt. Aus dieser Menge wird — mit Hilfe einer vorgegebenen Strategie — die letztendlich anzuwendende Regel ausgewählt.

An Strategien zur Regelauswahl gibt es nach Puppe ([Puppe 1988]):

- Auswahl nach Reihenfolge:
  - Die erste anwendbare Regel wird angewendet. Diese Trivialstrategie erlaubt eine Einflußnahme auf die Auswahl bei der Programmierung.
  - Die aktuellste Regel, d.h. die, deren Vorbedingungen sich auf die neuesten Fakten der Datenbasis beziehen, wird angewendet.
- Auswahl nach syntaktischer Struktur der Regel:
  - Die spezifischste Regel wird angewendet. Dies ist diejenige, deren Menge von Vorbedingungen eine Obermenge der Vorbedingungen anderer anwendbarer Regeln ist.
  - Die Regel mit den meisten Aussagen wird angewendet.
- Auswahl mittels Zusatzwissen:
  - Die Regel mit der größten Priorität — repräsentiert z.B. durch eine ihr zugeordnete Zahl — wird angewendet. Werden die Prioritäten statisch vergeben, so läßt sich dies auch durch die zuerst genannte Trivialstrategie erreichen.
  - Die Auswahl der anzuwendenden Regel wird durch Meta–Regeln gesteuert.

Prinzipiell wird dann, um Ableitungen durchzuführen, nach dem folgenden Algorithmus vorgegangen:

Algorithmus 2.2.1

Daten := *Ausgangsdatenbasis*
until Daten *erfüllt Terminierungskriterium* **do**
begin
    wähle anwendbare Regel R, der Vorbedingung durch Daten *erfüllt ist*
    Daten := *Ergebnis der Anwendung des Aktionsteils von R auf* Daten
end

Innerhalb der Aktionstheorie sollen Aussagen über die Entwicklung des Gegenstandsbereichs (bzw. der darin gültigen Aussagen) in der Vergangenheit und der Zukunft getroffen werden. Zwei der dabei interessierenden Fragestellungen sind die folgenden ([Shoham 1986]):

- Das *Frame Problem*: Welche Aussagen (Fakten) bleiben nach der Ausführung einer Aktion unverändert?
- Das *Qualification Problem*: Wann kann angenommen werden, daß eine Aktion erfolgreich ist?

Ein Rahmen zur Untersuchung von Aktionen ist der Situationskalkül (nach McCarthy und Hayes; hier: [Lifschitz 1987]). Er beschreibt Zustände[4] des Gegenstandbereichs als Ergebnis der

[4] Um eine Verwechslung mit der Situationstheorie (s.o.) zu vermeiden, ist hier und im folgenden von *Zuständen* statt von Situationen die Rede.

Ausführung von Aktionen. Ein Zustand z_0 geht durch die Ausführung einer Aktion a in einen Zustand z_1 über. Es gilt dann

$$z_1 = result(a, z_0).$$

Das Wissen über den Gegenstandsbereich in einem Zustand ist festgelegt durch die Gültigkeit von Relationen. Trifft im Zustand z_0 R zu, so gilt die (Meta–) Relation

$$holds(R, z_0).$$

Die Voraussetzungen der Ausführung und die Wirkung von Aktionen werden durch zwei weitere Relationen beschrieben. Damit eine Aktion ausgeführt werden kann, muß eine (evtl. leere) Menge von Vorbedingungen gültig sein. Diese Information wird durch

$$precond(R, a)$$

ausgedrückt, wenn a eine Aktion und R eine für ihre Ausführung notwendige Bedingung darstellt. Die Auswirkung von Aktionen sind Änderungen der Wahrheitswerte von Relationen; sie werden durch

$$causes(a, R, true)$$
$$causes(a, R, false)$$

beschrieben, wenn eine Relation R nach Ausführung der Aktion a wahr bzw. falsch wird.

Die genannten Aussagen bilden für jeden Zustand, jede Aktion und jede Relation eine Menge von Axiomen. Für eine formale Beschreibung der obengenannten Fragestellungen wird zunächst definiert:

$$success(a, z) \equiv \forall R(precond(R, a) \rightarrow holds(R, z))$$
$$affects(a, R, z) \equiv success(a, z) \wedge \exists v(causes(a, R, v))$$

Damit werden zwei Axiome definiert, die beschreiben, wie sich der Wert einer Relation nach Ausführung einer Aktion ändert bzw. nicht ändert:

$$success(a, z) \wedge causes(a, R, v) \rightarrow (holds(R, result(a, z)) \equiv v = true)$$
$$\neg affects(a, R, z) \rightarrow (holds(R, result(a, z)) \equiv holds(R, z)).$$

Mit Hilfe der minimalen Mengen von Tupeln, für die *causes* bzw. *precond* gelten (also genau die durch die entsprechenden Axiome festgelegten Tupel), lassen sich *Frame Problem* und *Qualification Problem* lösen.

Nachteil des Situationskalküls ist die hohe Komplexität, die die praktische Anwendbarkeit auf einfache Probleme beschränkt ([Pednault 1989]).

Eine Sprache zur formalen Beschreibung von Aktionen schlägt Pednault mit der *Action Description Language* (ADL) vor ([Pednault 1986], [Pednault 1989]). Auf diese soll im folgenden genauer eingegangen werden.

Action Description Language (ADL)

ADL vereinigt syntaktische Konzepte der Sprache STRIPS (nach Fikes und Nilsson) mit der Ausdrucksmächtigkeit des Situationskalküls. In ADL können Aktionen beschrieben werden, die abhängig sind vom Zustand des Weltausschnitts, in dem die Aktion ausgeführt wird.

Aktionen werden in ADL durch vier Gruppen von Klauseln beschrieben:

- Die *Vorbedingungen* (PRECOND). Hier werden die Relationen angegeben, die für die Ausführung der Aktion gelten müssen.
- Die *Liste der hinzuzufügenden Relationen* (ADD). Dies sind die Relationen, die durch die Ausführung der Aktion etabliert werden.
- Die *Liste der zu löschenden Relationen* (DELETE). Dies sind die Relationen, die durch die Ausführung der Aktion ungültig werden.
- Die *Liste der zu aktualisierenden Relationen* (UPDATE). Dies sind die Relationen, deren Argumente sich durch die Ausführung der Aktion ändern.

Aktionen zum Ablegen eines Objekts auf einem anderen und für eine einfache Zuweisung sollen die Syntax von ADL illustrieren (nach [Pednault 1986]):

$$
\begin{array}{lrcl}
Put(p,q) & PRECOND & : & p \neq q, p \neq TABLE, \forall z \neg On(z,p), (q = TABLE \vee \forall z \neg On(z,q)) \\
 & ADD & : & On(p,q) \\
 & DELETE & : & On(p,z)\ für\ alle\ z\ so\ daß\ z \neq q \\
 & & & \\
Assign(u,v) & UPDATE & : & Val(u) \longleftarrow Val(v)
\end{array}
$$

Die Zustandsübergänge für jedes einzelne Relationssymbol werden im Situationskalkül durch Axiome der folgenden Form beschrieben:

$$\forall x_1, \ldots, x_n, z\,(\pi^a(z) \wedge \alpha_R(x_1, \ldots, x_n, z) \rightarrow R(x_1, \ldots, x_n, result(a,z)))$$
$$\forall x_1, \ldots, x_n, z\,(\pi^a(z) \wedge \delta_R(x_1, \ldots, x_n, z) \rightarrow \neg R(x_1, \ldots, x_n, result(a,z)))$$

Dabei bezeichnet π^a die Vorbedingungen für die Ausführung der Aktion a. Das α_R–Axiom definiert die Bedingungen, unter denen $R(x_1, \ldots, x_n, result(a,z))$ wahr wird (d.h. $R(x_1, \ldots, x_n)$ wahr wird in dem Zustand, der durch die Ausführung der Aktion a erreicht wird), entsprechend ein δ_R–Axiom die Bedingungen, unter denen $R(x_1, \ldots, x_n, result(a,z))$ falsch wird. $\alpha_R(x_1, \ldots, x_n, z)$ ist die *Hinzufügebedingung (add condition)*, $\delta_R(x_1, \ldots, x_n, z)$ die *Löschbedingung (delete condition)* für das Symbol R.

Bei Funktionssymbolen haben die Zustandsübergangsaxiome die Form

$$\forall x_1, \ldots, x_n, y, z\,(\pi^a(z) \wedge \mu_F(x_1, \ldots, x_n, z) \rightarrow F(x_1, \ldots, x_n, result(a,z)) = y)$$

wobei μ_F definiert, wann $F(x_1, \ldots, x_n, result(a,z))$ der Wert y zugewiesen wird.

Eine Relation bzw. eine Funktion ändert ihren Wert durch die Ausführung einer Aktion nur, wenn dies explizit in der Aktionsbeschreibung festgelegt ist. Ist also z.B. $R(x_1, \ldots, x_n, z)$ wahr, so gilt auch $R(x_1, \ldots, x_n, result(a,z))$ noch, wenn nicht in der Definition von a ausdrücklich festgelegt ist, daß es nach der Ausführung nicht mehr gelten soll.

Es gilt demnach ($\overline{x}$ steht für $x_1, \ldots, x_n$)

$$\forall \overline{x}, z\,(\pi^a(z) \wedge R(\overline{x}, z) \wedge \neg\delta_R(\overline{x}, z) \rightarrow R(\overline{x}, result(a, z)))$$

$$\forall \overline{x}, z\,(\pi^a(z) \wedge \neg R(\overline{x}, z) \wedge \neg\alpha_R(\overline{x}, z) \rightarrow \neg R(\overline{x}, result(a, z)))$$

bzw.

$$\forall \overline{x}, z\,(\pi^a(z) \wedge \neg\exists y \mu_F(\overline{x}, y, z) \rightarrow F(\overline{x}, result(a, z)) = F(\overline{x}, z))$$

Damit sind die Wahrheitswerte der Relationen vollständig durch den vorherigen Wert und den Hinzufüge– bzw. Löschbedingungen festgelegt. Entsprechend ist auch der Wert einer Funktion nur von ihrem alten Wert und den Aktualisierungsbedingungen abhängig.

Jedem Literal in der ADD–Liste ist nun eine Hinzufügebedingung $\overset{\wedge}{\alpha}_R$ für ein Relationssymbol R zugeordnet. Ihre Disjunktion ergibt die Gesamtbedingung α_R. Ist keine entsprechende Bedingung spezifiziert, so gilt $\alpha_R \equiv false$. Man erhält α_R aus R folgendermaßen:

Relation	$\overset{\wedge}{\alpha}_R\ (x_1, \ldots, x_n)$
$R(\tau_1, \ldots, \tau_n)$	$(x_1 = \tau_1 \wedge \ldots \wedge x_n = \tau_n)$
$R(\tau_1, \ldots, \tau_n)$ wenn ψ	$(x_1 = \tau_1 \wedge \ldots \wedge x_n = \tau_n \wedge \psi)$
$R(\tau_1, \ldots, \tau_n)$ für alle $z_1, \ldots, z_k$	$\exists z_1, \ldots, z_k (x_1 = \tau_1 \wedge \ldots \wedge x_n = \tau_n)$
$R(\tau_1, \ldots, \tau_n)$ für alle $z_1, \ldots, z_k$, sodaß ψ	$\exists z_1, \ldots, z_k (x_1 = \tau_1 \wedge \ldots \wedge x_n = \tau_n \wedge \psi)$

Die logische Formel $\phi_R(x_1, \ldots, x_n)$, die aussagt, ob $R(x_1, \ldots, x_n)$ nach der Ausführung einer Aktion wahr ist, ist dann

$$\alpha_R(x_1, \ldots, x_n) \vee \neg\delta_R(x_1, \ldots, x_n) \wedge R(x_1, \ldots, x_n)$$

Aus der Formel wird die bereits erwähnte Tatsache ersichtlich, daß eine Relation in einem Zustand wahr ist, wenn sie entweder durch die ausgeführte Aktion wahr wird oder bereits vorher wahr ist und durch die Aktion nicht falsch wird.

Beispiel 2.2.1 *Hat die Definition einer Aktion die Form*

$$\begin{aligned} PRECOND &: \ldots \\ ADD &: R(s, t) \\ DELETE &: R(u, v) \end{aligned}$$

so ist die Gültigkeit des Fakts $R(x_1, x_2)$ nach ihrer Ausführung durch

$$\varphi_R(x_1, x_2) \equiv (x_1 = t \wedge x_2 = s) \vee (R(x_1, x_2) \wedge \neg(x_1 = u \wedge x_2 = v))$$

gegeben.

Entsprechendes gilt auch für δ– und μ–Bedingungen; dies ist in [Pednault 1989] ausgeführt.

2.2.3 Bezüge von SUSI zur Situations– und Aktionstheorie

Beim Entwurf von SUSI wurden die in den vorigen beiden Abschnitten eingeführten Konzepte verwendet. Dabei orientiert sich hauptsächlich die Systemstruktur und die Art der Wissensrepräsentation an den Elementen der Situationstheorie. Für das Ausführungsmodell werden die in der Aktionstheorie konzipierten Aktionsregeln verwendet. Wie diese theoretischen Elemente in das SUSI–Konzept eingeflossen sind, wird in diesem Abschnitt beschrieben.

Situationen

Im Abschnitt 2.2.1 wurden die Strukturen der Wissensdarstellung in der Situationstheorie skizziert. Die genannten Strukturelemente sind in die Grundkonzeption von SUSI eingegangen; ihre Bedeutung ist allerdings teilweise eine andere als in der originalen Situationstheorie.

Grundelement ist in beiden Fällen der Begriff der (abstrakten) Situation als Beschreibung eines Ausschnitts aus der Welt. Dieser Ausschnitt wird durch eine Menge von Fakten beschrieben; weitere Information resultiert aus der Anwendung von Hintergrundwissen auf diese Fakten. Dieses Hintergrundwissen wird in SUSI durch Nebenbedingungen explizit beschrieben, die z.B. als Formeln der Prädikatenlogik erster Stufe formalisiert werden können.

Wenn jedoch eine Situation in der zugrundeliegenden Theorie auch zeitgebunden ist, d.h. mit fortschreitender Zeit ständig neue Situationen beobachtet werden, verstehen wir unter einer Situation ein abstraktes Gebilde, das sich mit der Zeit verändern kann. Daraus ergeben sich dann auch einige Änderungen in der Bedeutung der anderen Konstrukte.

In Situationstypen können die im zeitlichen Verlauf beobachteten Situationen gemäß der innerhalb eines Beobachtungszeitraums unveränderlichen Fakten zusammengefaßt werden. In SUSI spricht man hier immer von derselben Situation, deren innere Struktur (d.h. die in ihr enthaltene, durch Fakten festgehaltene Information) sich laufend ändert.

Im Gegensatz zur ursprünglichen Theorie, in der Situationstypen nach Maßgabe existierender Gemeinsamkeiten frei definiert werden können, besitzt SUSI ein festes Typkonzept: Situationen werden nicht nach in Fakten repräsentierten, gemeinsamen Informationen einzelnen Typen zugeordnet, sondern nach dem in den Nebenbedingungen festgelegten (abstrakten) Hintergrundwissen (und dem in Aktionsregeln beschriebenen möglichen Verhalten, s.u.). Die Ausprägungen eines bestimmten Typs haben das dynamische Verhalten und die komplexen Zusammenhänge gemeinsam; sie haben den gleichen Situationstyp und unterscheiden sich nur in ihren Fakten, die den konkreten Zustand repräsentieren.

Da Nebenbedingungen damit einem Situationstyp zugeordnet sind, haben auch sie in dem hier beschriebenen Konzept eine andere Bedeutung.

In der Situationstheorie stellen Nebenbedingungen Beziehungen zwischen Situationstypen dar — eine Aussage $S_0 \Rightarrow S_1$ bedeutet, daß, wenn eine Situation des Typs S_0 beobachtet wird, immer auch eine Situation des Typs S_1 beobachtet wird. In SUSI repräsentieren sie einfach Relationen zwischen Fakten innerhalb einer Situation. Zu beachten ist, daß die Bedeutung auf einer abstrakten Ebene die gleiche sein kann: an Stelle der Aussage „beobachte ich eine Situation des Typs A (welche das Fakt a enthält), so beobachte ich auch eine Situation B (mit dem Fakt b)“ tritt die Aussage „in einer Situation A beobachte ich, wenn ich a beobachte, auch b“. Daß das Fakt a das Fakt b impliziert, gilt in beiden Fällen. Auf der formalen Ebene unterscheiden sich die beiden Sichtweisen jedoch deutlich.

Die Fakten werden in SUSI die Form von Literalen annehmen. Das Fakt $\ll R, a_1, \ldots, a_n, 1 \gg$ aus der Situationstheorie hat in SUSI die Form $R(a_1, \ldots, a_n)$, $\ll R, a_1, \ldots, a_n, 0 \gg$ entsprechend

die Form $\neg R(a_1, \ldots, a_n)$. Unter der *Closed World Assumption* werden negative Fakten nicht repräsentiert; es gilt in dem Fall bei einer Faktenmenge $\mathcal{F}$ die Aussage $\mathcal{F} \models \neg R(a_1, \ldots, a_n)$ (bzw. $\mathcal{F} \models\ll R, a_1, \ldots, a_n, 0 \gg$), wenn $R(a_1, \ldots, a_n) \notin \mathcal{F}$ ($\ll R, a_1, \ldots, a_n, 1 \gg \notin \mathcal{F}$).

Aktionen

Durch Aktionen wird die dynamische (prozedurale) Seite einer SUSI–Spezifikation beschrieben. Wie in der herkömmlichen Aktionstheorie wird durch Regeln eine Wissensbasis manipuliert. Ziel ist es dabei jedoch zunächst nicht — wie meistens bei der Untersuchung von Aktionen — Pläne zur Erreichung eines gegebenen Ziels zu berechnen, sondern die Aktionen stellen *Prozeduren* dar, die zur Laufzeit ausgeführt werden und — neben der Manipulation der Wissensbasis — die umgebenden Systemkomponenten Präsentationsschicht und Anwendung — durch Seiteneffekte — steuern.

Den einzelnen Situationstypen sind in SUSI Aktionsregeln zugeordnet, die in den Situationen eines Typs einen Zustand — repräsentiert durch die gerade gültigen Fakten — in einen anderen überführen. Zusätzlich werden durch spezielle (prozedurale) Relationen Seiteneffekte auf andere Systemteile erzeugt. (Intern verhalten sich diese Relationen wie normale logische Relationen, d.h. sie werden ausgewertet und haben entsprechende Ergebnisse.)

Angestoßen wird die Ausführung einer Aktionsregel durch einen *Trigger*, einer Relation, die ein Ereignis repräsentiert, daß entweder von einer Situation oder von Präsentations– bzw. Anwendungsebene abgesetzt wurde. Dieser ist aus logischer Sicht nicht anderes als ein weiteres Fakt, das der Faktenmenge (temporär) hinzugefügt wird.

Eine Beschreibung von SUSI sieht allgemein folgendermaßen aus (Nebenbedingungen werden hier weggelassen; sie können als komplexe Fakten aufgefaßt werden. Kommata sind als Konjunktion zu lesen.):

$$
\begin{array}{ll}
A_1 & a_1^t, a_{11}, \ldots, a_{1n_1} \longrightarrow b_{11}, \ldots, b_{1m_1}, \neg c_{11}, \ldots, \neg c_{1l_1}, u_{11}^t, \ldots, u_{1r_1}^t \\
 & \ldots \\
A_k & a_k^t, a_{k1}, \ldots, a_{kn_k} \longrightarrow b_{k1}, \ldots, b_{km_k}, \neg c_{k1}, \ldots, \neg c_{kl_k}, u_{k1}^t, \ldots, u_{kr_k}^t \\
 & d_1 \\
 & \ldots \\
 & d_s
\end{array}
$$

Die Regel besteht dabei aus folgenden Teilen:

$$
\underbrace{A_i}_{Name}\ \underbrace{a_i^t}_{Trigger}, \underbrace{a_{i1}, \ldots, a_{in_i}}_{Vorbed.} \quad \longrightarrow \quad \underbrace{b_{i1}, \ldots, b_{im_i}}_{pos.\ Nachbed.}, \underbrace{\neg c_{i1}, \ldots, \neg c_{il_i}}_{neg.\ Nachbed.}, \underbrace{u_{i1}^t, \ldots, u_{ij_i}^t}_{neue\ Trigger}
$$

Wird nun ein Trigger v^t erzeugt, so wird — nach Auswahl einer Aktionsregel A — eine Funktion $\delta : F \times A \times v^t \longmapsto F$ angewendet, deren Resultat folgendermaßen definiert ist (die neuen Trigger sind hier nicht berücksichtigt):

$$
\delta(F, A, v^t) = \begin{cases} F \cup \{b_{i1}, \ldots, b_{im_i}\} \setminus \{c_{i1}, \ldots, c_{il_i}\} & : \quad F \cup \{v^t\} \models a_i^t \wedge a_{i1} \wedge \ldots \wedge a_{in_i} \\ F & : \quad sonst \end{cases}
$$

Da von einer geschlossenen Welt ausgegangen wird, entspricht das Hinzufügen eines Fakts zur Faktenbasis dem Hinzufügen eines Tupels zur Interpretation der jeweiligen Relation.

Wie oben bereits erwähnt, ist der Trigger v^t dabei aus logischer Sicht lediglich ein der Faktenbasis temporär zugefügtes Fakt (dabei wird o.B.d.A. davon ausgegangen, daß die Menge der Triggerrelationen und die der anderen Relationen disjunkt ist). Gleiches gilt für die neuen Trigger, die dann einfach den positiven Nachbedingungen zugeschlagen werden (sie würden im ersten Fall der Funktion δ mit der Menge F vereinigt). Zusätzlich besitzen die Trigger jedoch eine wichtige Bedeutung im Zusammenhang mit der Auswahl einer Aktionsregel.

Der temporäre Charakter der Trigger kann so interpretiert werden, daß der anstoßende Trigger einer Regel bei deren Ausführung wieder gelöscht wird, d.h. als zu löschende Relation in die Nachbedingung aufgenommen wird. Die Trigger der Nachbedingung sind aus dieser Sicht gewöhnliche (positive) Literale, die ebenfalls den Faktenbasen zugeschlagen werden. (Hier wird jedoch davon abgesehen, daß sie in einer durch die Regel vorgegebene Reihenfolge berücksichtigt werden.) Die besondere Stellung der Trigger erfordert dann, daß die Menge der (möglichen) Trigger–Relationen und die Menge der „normalen“ Relationen disjunkt sind.

Aus dieser Sicht entspricht eine SUSI–Regel

$$a\ trig(f(X)), g(X,Y), h(Z) \longrightarrow m(Y), not(n(Z,X)), trig(p(Z))$$

folgender Regel in ADL:

$$\begin{aligned} PRECOND &: f(X), g(X,Y), h(Z) \\ ADD &: m(Y), p(Z) \\ DELETE &: n(Z,X), f(X) \end{aligned}$$

Die Variablen X, Y, Z müssen dabei bei der Auswertung der Vorbedingung instantiiert werden.

Zusätzlich zu dem beschriebenen Aktionskonzept sind in SUSI zwei weitere Eigenschaften vorgesehen:

- Aufgrund der Modularisierung in Situationen existiert nicht eine Faktenbasis, sondern mehrere. Dabei können Trigger auch an andere Situationen gesendet, bzw. Vorbedingungen auch in anderen Situationen bewiesen werden.
- Zusätzlich interagiert der Interpreter mit der Anwendung und der Präsentationsebene des Systems. Wird aus diesen Bereichen Information benötigt, so werden sie als weitere (virtuelle) Situationen behandelt. *Prozedurale Relationen*, die Operationen in diesen beiden Systemkomponenten auslösen, werden durch Seiteneffekte — beispielsweise Funktionsaufrufe — realisiert.

Damit sind in der Vorbedingung einer Aktionsregel — zusätzlich zu den Situationsinternen Relationen $\pi_1, \ldots, \pi_n$ — weitere, aus den Situationen s_i stammende Relationen enthalten. Damit erhält die Vorbedingung die Form

$$\pi_1, \ldots, \pi_n, \pi^{s_1}{}_1, \ldots, \pi^{s_1}{}_m, \ldots, \pi^{s_k}{}_1, \ldots, \pi^{s_k}{}_l$$

In Abwandlung der Darstellung in [Puppe 1988] zeigt Abbildung 2.5 die Struktur einer Regel in SUSI (siehe dazu auch Abbildung 4.1 auf Seite 46). Eine weitergehende Betrachtung von SUSI, insbesondere auch von Aspekten der Realisierung, findet sich in den Kapiteln 4 und 5.

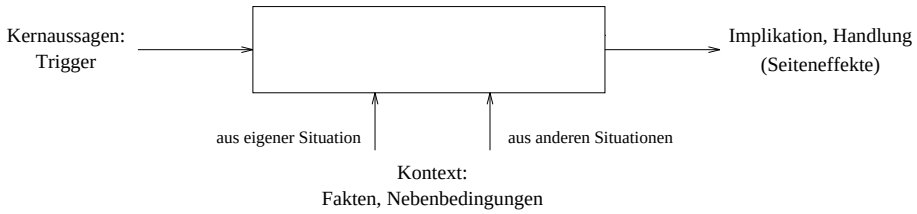

Abbildung 2.5: *Struktur einer Regel in* SUSI*: Den Anstoß gibt ein Trigger; anschließend werden Bedingungen aus dem — als Fakten und Nebenbedingungen repräsentierten — Kontext abgeleitet und ggf. Implikationen etabliert bzw. Handlungen ausgeführt.*

2.3 Zusammenfassung

In diesem Kapitel wurde in die beiden wesentlichen Aspekte der vorliegenden Arbeit eingeführt:

- Zunächst wurde der Bereich der graphischen Benutzungsschnittstellen dargestellt. Dabei wurde zunächst auf die Hauptanforderungen eingegangen und dann einige bestehende Grundprinzipien für die Realisierung beschrieben.

  Die Struktur graphischer Benutzungsschnittstellen zeigt drei wesentliche Komponenten: die Präsentationsschicht, also die sichtbare Oberfläche, die Dialogsteuerung und die Schnittstelle zur Anwendung. Es zeigt sich jedoch, daß die (aus der Sicht der Softwaretechnik wünschenswerte) Trennung der Teile nicht strikt aufrechterhalten werden kann. Auf semantische Information der Anwendung, von der die Darstellung abhängt, muß von der Präsentationsschicht aus beispielsweise zugegriffen werden können.

  Die ereignisbasierten Modelle, die heute weit verbreitet sind, können diesen Anforderungen nicht vollständig genügen. Eine Erweiterung dieses Konzepts stellen die produktionsbasierten Modelle dar, die mit Hilfe von Regeln (anstatt simpler Tabellen) die Benutzungsschnittstelle steuern. Hier kann auch das benötigte Anwendungswissen in der Dialogsteuerungskomponente explizit gemacht werden.

- Aus theoretischer Sicht basiert SUSI auf zwei Gebieten: der *Situationstheorie*, die die Struktur der Modellierung vorgibt und der *Aktionstheorie*, mit deren Hilfe das Vorgehen zur Laufzeit beschrieben wird. In diese beiden Gebiete wurde eingeführt und die Bezüge des Ansatzes von SUSI auf diese Gebiete skizziert.

  Die Strukturelemente — Situationen und Situationstypen —, die aus der Situationstheorie entnommen sind, dienen zur Modularisierung von Beschreibungen der Benutzungsschnittstellen. Zusätzlich werden durch Nebenbedingungen komplexe Zusammenhänge (Hintergrundwissen) dargestellt.

  Diese Struktur wird mit Aktionen (wie sie auch bei anderen produktionsbasierten Systemen vorhanden sind) zusammengeführt, durch die das dynamische Verhalten beschrieben werden kann. Sie entsprechen weitgehend den aus der Aktionstheorie bekannten.

Aufbauend auf diesen Grundlagen soll im Kapitel 4 der Ansatz von SUSI entwickelt werden. Zunächst werden im folgenden Kapitel noch einige bereits bestehende verwandte Ansätze dargestellt.

3

Existierende Ansätze

In diesem Kapitel werden einige bereits existierende Ansätze vorgestellt, die alle eine Dialogsteuerung mit Hilfe von Regeln vornehmen und insofern eine Verwandtschaft zu dem von uns gewählten Ansatz aufweisen.

Nach der in [Olsen 1992] vorgenommenen Einteilung handelt es sich dabei um *Produktionssysteme*, die entwickelt wurden, nachdem deutlich wurde, daß zustandsbasierte Automaten für die Aufgabenstellung nur bedingt geeignet sind; insbesondere wegen der (z.B. im Fall von beliebiger Eingabereihenfolge) exponentiell wachsenden Anzahl von notwendigen Zuständen und Übergängen (gemessen an der Zahl der verschiedenen Eingabemöglichkeiten). Ein weiterer Aspekt ist die gegenüber tabellengesteuerten ereignisbasierten Systemen verbesserte Möglichkeit der internen Informationsdarstellung.

Zunächst wird dabei auf das Mitte der achziger Jahre in Toronto (Kanada) entwickelte System SASSAFRAS eingegangen. In diesem Zusammenhang interessiert insbesondere die *Event–Response–Language* (ERL), eine Sprache zur Beschreibung des Mensch–Maschine–Dialogs mit Hilfe von Regeln, die für SASSAFRAS entwickelt wurde. Auch das dazugehörige Architektur– und Kommunikationsmodell, die *Local Event Broadcast Method* (LEBM) wird dabei kurz behandelt ([Hill 1986]).

Eine Obermenge von ERL ist das *Propositional Production System* (PPS), das in Provo (Utah, USA) entwickelt wurde. Es verbindet regelbasierte Konzepte mit zustandsbasierten Ansätzen ([Olsen], [Olsen 1992]). Grundlage sind dabei *Zustandsvektoren* mit deren Hilfe die auszuführenden Regeln ausgewählt werden.

Als letztes regelbasiertes System betrachten wir das am Georgia Institute of Technology (USA) — teilweise in Kooperation mit der Technischen Universität Delft (Niederlande) — entwickelte *User Interface Design Environment* (UIDE), in dem ebenfalls, ähnlich wie in SUSI, der Benutzerdialog mit Hilfe von Vor– und Nachbedingungen gesteuert wird, die einzelnen Bildschirmobjekten, wie z.B. Knöpfen, zugeordnet sind ([Foley et al. 1989], [Gieskens, Foley 1991], [Sukaviriya et al. 1993]). Dieses System ist hier auch deswegen von besonderem Interesse, weil im späteren Verlauf des Projekts ein kontextorientiertes Hilfssystem entwickelt wurde, das die in den Regeln dargestellten Informationen zusammen mit der aktuellen Zustandsinformation verwendet, um Hilfstexte zu generieren ([de Graaff 1992], [de Graaff et al. 1993]).

3.1 Sassafras

SASSAFRAS ([Hill 1986]) ist ein Benutzungsschnittstellen–Entwicklungssystem (User Interface Development System, UIDS), das Mitte der achtziger Jahre entwickelt wurde. Ziele bei seiner Entwicklung waren vor allem:

- Unterstützung parallel ablaufender Dialoge
- Bereitstellung eines Mechanismus zur Kommunikation zwischen den Modulen, die die Dialoge unterstützen
- Synchronisation dieser Dialoge und Konsistenz des Systemverhaltens

Die beiden wesentlichen Neuerungen bei SASSAFRAS, die die genannten Anforderungen erfüllen sollen, sind:

- die *Event–Reponse–Language* (ERL) zur Dialogbeschreibung,
- die *Local Event Broadcast Method* (LEBM) als Architekturansatz.

Im Hinblick auf die vorliegende Arbeit ist vor allem ERL, in der Benutzerdialoge mit Hilfe von Produktionen (ähnlich dem in SUSI verwendeten Konzepts der Aktionsregeln) beschrieben werden von Interesse. Deswegen soll im folgenden darauf etwas genauer eingegangen werden. Danach folgt eine kurze Erläuterung von LEBM.

3.1.1 Event–Response–Language

Die Sprache wurde entwickelt, um die syntaktische Ebene des Mensch–Maschine–Dialogs zu beschreiben. Ihre Hauptbestandteile sind *Ereignisse* und *Flags*. Dabei werden erstere unterschieden nach *Eingangs*ereignissen und *Ausgangs*ereignissen. Ereignisse sind zunächst Signale, die anzeigen, daß etwas passiert ist; sie können dabei Daten enthalten (z.B. kann ein Mausklick auf eine bestimmte Stelle ein Ereignis erzeugen; die mitgeführten Daten beschreiben die dazughörigen Koordinaten (oder auch das angeklickte Präsentationsobjekt)). Sie sind die einzige Form der Ein– bzw. Ausgabe von ERL–Modulen. Flags repräsentieren als lokale Variablen den Zustand des Systems.

Eine Dialogspezifikation in ERL besteht aus einer Menge von Regeln der Form $Bedingung \longrightarrow Aktion$. Dabei kann die *Bedingung* aus einem Ereignis und einer Liste von Flags bestehen, oder nur aus einer Liste von Flags (sogenannte ϵ–Regeln). Die Aktion setzt sich zusammen aus neu zu setzenden Flags, neuen Ereignissen und Zuweisungen.

Eine Regel ist *anwendbar*, wenn

- an der Spitze der Eingabewarteschlange das Ereignis aus der Bedingung steht (diese Voraussetzung entfällt bei ϵ–Regeln) und
- alle Flags in der Bedingung gesetzt sind.

Wird die Regel angewendet, werden zunächst alle Flags in der Bedingung zurückgesetzt, dann (in der Reihenfolge des Auftretens) die Flags in der Aktion gesetzt, die Ereignisse versendet (die Verteilung übernimmt LEBM) und die Zuweisungen ausgeführt.

Zur Illustration soll das folgende Beispiel dienen, das (in einem Zeichenprogramm) den Beginn des Zeichnens einer Linie beschreibt:

Beispiel 3.1.1

```
MouseDown waitingForLine ⟶
StartLine.X ← MouseDown.X
StartLine.Y ← MouseDown.Y
StartLine !
lineDragging ↑
```

`MouseDown` ist dabei das Ereignis des Mausklicks; dabei werden die Parameter `X` und `Y` (die Bildschirmkoordinaten) übermittelt. Das Flag `waitingForLine` ist gesetzt in dem Systemzustand, in den eine Linie gezeichnet werden soll. Es wird dann das Ereignis `StartLine` an den Kommunikationsmechanismus gesendet; dabei werden die Mauskoordinaten als Startwerte der Linie mitgegeben. Zuletzt wird das Flag `lineDragging` gesetzt, um den neuen Systemzustand anzuzeigen.

In Abbildung 3.1 wird als größeres Beispiel der beim Einloggen in ein System auftretende Dialog beschrieben (aus [Hill 1986]).

Zu Beginn wird eine Systemmeldung ausgegeben und ein Flag gesetzt, daß der *Login*–Name erwartet wird. Die zweite Regel ist ein Beispiel für eine ϵ–Regel; sie beschreibt diesen Eingabeprozeß. Die dritte Regel wird aktiviert, wenn eine Eingabe gemacht wurde (das Ereignis `GotPromptWord`) während ein *Login*–Name erwartet wird (das Flag `waitUserid`). Das System erwartet dann die Eingabe des Passworts.

Anschließend wird die Richtigkeit des Passworts geprüft und, je nach Ergebnis, der Erfolg festgestellt, die Eingabe wiederholt oder — nach zu vielen erfolglosen Eingaben — der Vorgang abgebrochen.

3.1.2 Local Event Broadcast Method

Die Local Event Broadcast Method umfaßt Architektur und Kommunikationsmechanismus von Sassafras. Sie gruppiert *Module* (logische Einheiten einer Benutzungsschnittstelle, diese können z.B. in ERL oder auch in Lisp geschrieben sein) zu *Clustern*. Die Module innerhalb eines Clusters kommunizieren durch das Senden von Ereignissen; dies wird durch einen *Cluster–Controller* überwacht. Dieser verwaltet zwei Warteschlangen in der gesendete Ereignisse eingereiht werden; eine für Ereignisse innerhalb und eine für Ereignisse außerhalb eines Clusters. Dabei wird die interne Schlange vorrangig abgearbeitet.

Bei einfacheren Benutzungsoberflächen umfaßt ein Cluster alle Komponenten; komplexere Systeme können auch in mehrere Cluster zerlegt werden. Den Aufbau von einer Oberflächenspezifikation von Sassafras mit LEBM zeigt Abbildung 3.3 (nach [Hill 1986]).

```
INITIALLY ⟶
     TtyOut !
         .printout ← ‘‘Welcome to angluin’’
     getUserid ↑

--- getUserid ⟶
     PromptForWord !
         .prompt ← ‘‘login: ’’
         .echoForm ← NIL
     waitUserid ↑

GotPromptWord waitUserid ⟶
     CheckLogin !
         .userid ← GotPromptWord.input
     PromptForWord !
         .prompt ← ‘‘Password: ’’
         .echoForm ← ‘‘*’’
     waitPassword ↑

GotPromptWord waitPassword ⟶
     CheckLogin !
         .password ← GotPromptWord.input

GoodLogin ⟶
     LoginSuccessful !

BadLogin ⟶
     TtyOut !
         .printout ← ‘‘Login incorrect’’
     getUserid ↑

TooManyAttempts ⟶
     TtyOut !
         .printout ← ‘‘Too many attempts’’
     LoginUnSuccessful !
```

Abbildung 3.1: *Beispiel für eine ERL–Spezifikation (vereinfacht)*

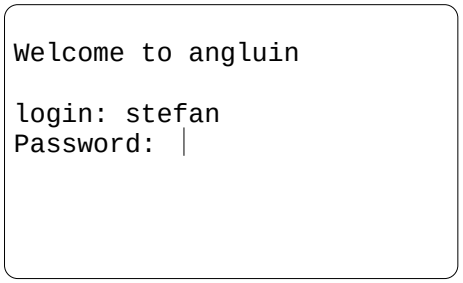

Abbildung 3.2: *Die Benutzersicht*

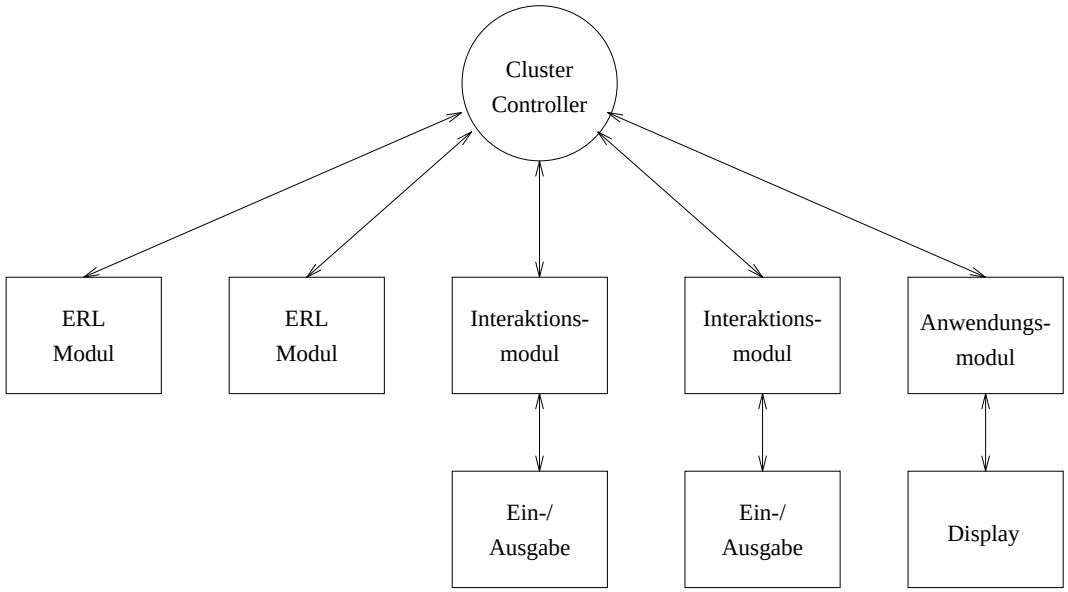

Abbildung 3.3: *Architektur einer mit* SASSAFRAS *implementierten Benutzungsoberfläche.*

3.2 Propositional Production System (PPS)

Anfang der neunziger Jahre wurde das *Propositional Production System* (PPS) vorgestellt, eine Obermenge der zustandsbasierten Systeme, der Ereignistabellen und der Event–Response–Systeme ([Olsen], [Olsen 1992]). Auch dieses System realisiert eine regelbasierte Steuerung von graphischen Benutzungsoberflächen durch Aktionsregeln. Es zeichnet sich vor allem durch die folgenden Eigenschaften aus:

- Es läßt sich aus dem Systemzustand (repräsentiert durch einen Vektor von Bedingungen) jederzeit ermitteln, welche Eingaben an der Benutzungsoberfläche aktuell möglich sind.
- Es werden semantische Eigenschaften von Objekten unterstützt; z.B. kann ein Sicherungsbefehl nicht ausgeführt werden, wenn keine Datei geöffnet ist.
- Es existiert ein Vererbungsmechanismus, d.h. das Verhalten anderer PPS–Definitionen kann geerbt werden.

Darauf, wie diese Eigenschaften, die sich für die Beschreibung gerade von graphischen Benutzungsschnittstellen als wesentlich erweisen, realisiert sind, soll in diesem Abschnitt beschrieben werden.

3.2.1 Informelle Beschreibung eines PPS

Ein PPS besteht aus einer Zustandsraumdefinition und einer Menge von Regeln. Ein Zustandsraum entspricht einer Menge von Bedingungen, vergleichbar mit Ereignissen in Event–Response–Systemen. Bedingungen werden in Feldern zusammengefaßt; dabei schließen sich innerhalb eines Feldes die verschiedenen Bedingungen gegenseitig aus. In einem Zeichenprogramm kann ein Zustandsraum etwa so aussehen:

Beispiel 3.2.1 (Zustandsraumdefinition in PPS)

```
Field ActiveCommand (NoCommand, LineCommand, CircleCommand,
RectangleCommand)

Semantic Field SelectedObject (NoSelection, LineSelection,
CircleSelection, RectangleSelection)

Field DragMode (RubberLine, RubberCircle, RubberRectangle,
DragSelect, NotDragging)

Query Field ConfirmDelete (OKDelete, CancelDelete)

Input Field Input (NullEvent, MouseDown, MouseUp, LineMenu,
CircleMenu, RectangleMenu, DeleteMenu)

Field Action (DrawLine, DrawRectangle, DrawCircle, SaveMousePoint,
DeleteLine, DeleteRectangle, DeleteCircle)
```

Die einzelnen Felder enthalten zwei oder mehr Bedingungen, deren Namen innerhalb des PPS eindeutig sein müssen. Sie sind nach ihrer Bedeutung in vier Kategorien eingeteilt:

- Zustandsinformation (im obigen Beispiel `ActiveCommand`, `DragMode`, `Action`): Die entsprechenden Bedingungen werden aus dem aktuellen Zustandsvektor gelesen.
- Eingabeinformation (`Input`): Diese beschreibt die Eingabe an der Benutzungsoberfläche.
- Semantische Bedingungen (`SelectedObject`): Sie definieren die von der Anwendung auszuführenden semantischen Aktionen.
- Abfragen (`ConfirmDelete`): Durch sie wird beispielsweise die Bestätigung für eine angestoßene Aktion angefordert.

Die genannten Bedingungen werden, wie oben im Beispiel zu sehen, alle einheitlich als Felder im Zustandsraum dargestellt.

Auf dem Zustandsraum werden dann Regeln definiert, die Zustände ineinander überführen. Beispielhaft sind hier einige dieser Regeln dargestellt:

Beispiel 3.2.2 (PPS–Regeln)

```
NoCommand, LineMenu ⟶ LineCommand, NotDragging

NoCommand, CircleMenu ⟶ CircleCommand, NotDragging

NoCommand, RectangleMenu ⟶ RectangleCommand, NotDragging

LineCommand, MouseDown ⟶ RubberLine, SaveMousePoint

RubberLine, NullEvent ⟶ RubberLine

RubberLine, MouseUp ⟶ NotDragging, DrawLine

DeleteMenu, LineSelect, OKDelete ⟶ DeleteLine, NoCommand

DeleteMenu, RectangleSelect, OKDelete ⟶ DeleteRectangle, NoCommand

DeleteMenu, CircleSelect, OKDelete ⟶ DeleteCircle, NoCommand
```

Ausgeführt werden kann eine solche Regel, wenn die Bedingungen auf der linken Seite im aktuellen Zustand (einem Vektor aus den oben bereits erwähnten Zustandsraum) enthalten sind. Alle nicht vorhandenen Eigenschaften werden als *„don't care"*–Elemente behandelt; sie spielen bei der Auswahl der anzuwendenden Regeln keine Rolle.

Beispiel 3.2.3 *Ist der aktuelle Bedingungsvektor von der Form* `(LineCommand, MouseDown)`*, so sind alle Regeln, deren linke Seite diese Bedingungen enthalten, ausführbar, z.B.* `LineCommand, NoSelection, MouseDown, DrawLine --> ....`

Semantische Aktionen, Eingabe, Abfragen

Semantische Aktionen werden mit Bedingungsfeldern und ihren Elementen aus dem Zustandsraum assoziiert. Dabei werden in Bedingungen aus einem Feld mit der Bezeichnung `Semantic Field` ausschließlich semantische Aktionen definiert; in anderen Feldern können semantische Aktionen zur Dialogkontrolle hinzutreten. Die Form semantischer Aktionen ist vielfältig; typischerweise handelt es sich z.B. um Adressen von LISP– oder C–Routinen.

Die Regeln eines PPS werden bei jeder neuen Eingabe ausgeführt. Entsprechend der Eingabebedingungen (festgelegt im `Input Field`) werden Eingaben gelesen (dem aktuellen Zustandsvektor hinzugefügt) und dann das PPS ausgewertet. Zusätzlich müssen zur Aktivierung bzw. Deaktivierung von Eingabemöglichkeiten (Logischen Einheiten wie etwa Menüelementen) noch Informationen herangezogen werden: weisen alle Regeln, die in ihrem Bedingungsvektor eine bestimmte Eingabe voraussetzen, Widersprüche zum aktuellen Zustandsvektor in einem oder mehreren Elementen auf, so ist klar, daß diese Regeln nicht ausgeführt werden können, die Eingabe also im momentanen Kontext keinen Sinn hat. Ist dies der Fall, so sollte die logische Eingabeeinheit deaktiviert werden, z.B. durch graues Unterlegen.

Abfragefelder, die z.B. Sicherheitsabfragen (*„OK–Buttons"*) realisieren, sind Bedingungen, zu deren Auswertung eine Abfrageaktion (wie das Öffnen eines Menü–Fensters mit den Wahlmöglichkeiten `OK` und `Cancel`) ausgeführt, und die Bedingung gemäß der darauf erfolgenden Eingabe gesetzt wird.

3.2.2 Formale Definitionen

Um die Vorgehensweise bei der Auswertung eines PPS zu beschreiben, benötigen wir zunächst noch einige Definitionen. Dabei wird von der folgenden Zustandsraumdefinition ausgegangen:

```
Field X (A, B, C)
Field Y (P, Q)
Field Z (T, U, V)
```

Wie bereits erwähnt, ist der *Bedingungsvektor* das zentrale Element eines PPS. Er wird durch eine Liste von Bedingungen repräsentiert, wobei von einem Feld maximal eine Bedingung gültig sein kann. Ist aus einem Feld keine Bedingung im Bedingungsvektor enthalten, so wird dies als *„don't care"*–Belegung interpretiert, d.h. das Feld wird nicht berücksichtigt.

Beispiel 3.2.4 *Der Bedingungsvektor* `(A, T)` *repräsentiert im oben angegebenen Zustandsraum die Zustandsmenge* { `(A, P, T), (A, Q, T)` }.

Auf diesen Vektoren werden einige Operationen definiert:

> Definition 3.2.1 (Untermenge) *Der Vektor* a *ist eine Untermenge des Vektors* b *genau dann, wenn jede Bedingung aus* b *auch in* a *enthalten ist.*

Zum Beispiel ist `(B, T)` Untermenge von `(B)`, aber nicht von `(B, Q)`.

> Definition 3.2.2 (Schnittmenge) *Es ist* r = a ∩ b *genau dann, wenn jede Bedingung aus* a *oder aus* b *in* r *enthalten ist und in* r *nicht mehr als eine Bedingung in einem Feld enthalten ist.*

Zum Beispiel gilt `(B, T)` ∩ `(Q, T)` = `(B, Q, T)` und `(B, T)` ∩ `(Q, U)` = `()`.

> Definition 3.2.3 (Überlagerung) *Es ist* r = a *überlagert* b *genau dann, wenn:*
>
> *Ist ein Feld in* a *ein* don't care*, dann erhält das entsprechende Feld in* r *den Wert aus* b*; sonst erhält es den Wert von* a*.*

Das Ergebnis ist also b, bei dem die relevanten Felder so geändert sind, daß sie auf a passen.

Zwei Bedingungsvektoren heißen *unabhängig* genau dann, wenn

$$\text{a } \textit{überlagert} \text{ b} = \text{b } \textit{überlagert} \text{ a},$$

d.h. die Bedingungen sind entweder gleich oder aus verschiedenen Feldern. Zwei Regeln erzeugen einen *Konflikt*, wenn die linken Seiten eine nichtleere Schnittmenge haben (ein Zustand existiert, in dem beide anwendbar sind) und die rechten Seiten nicht unabhängig sind (Felder auf einander widersprechende Bedingungen gesetzt werden).

3.2.3 Auswertung eines PPS

Die Auswertung eines PPS geschieht folgendermaßen: Zur Laufzeit werden Ausprägungen eines PPS erzeugt. Diese enthalten je einen Zeiger auf die entsprechende PPS–Definition und den aktuellen Zustand (*current state*, CS). Zur Auflösung von Konflikten werden die bekannten Bedingungen in einem zusätzlichen Bedingungsvektor (*known conditions*, KC) repräsentiert, der vor der Auswertung dem aktuellen Zustand überlagert wird. Bei einer Eingabe wird KC auf die Eingabebedingungen gesetzt, die Eingabefelder von CS erhalten die durch den Eingabevorgang festgelegten Werte, und anschließend wird das PPS ausgewertet. Der prinzipielle Algorithmus ist dabei der folgende:

> Algorithmus 3.2.1 (PPS–Auswertung für eine einzelne Eingabe)
>
> *CS := Ergebnis der davorliegenden Eingabe*
> *KC := Eingabebedingungen aus dem letzten Eingabeereignis. Andere Felder* don't care.
> *CS := KC überlagert CS*
> *Für jede Regel R (R.links ⟶ R.rechts) von höchster bis niedrigster Priorität*
> *Wenn CS ist Untermenge von R.links und R.rechts ist von KC unabhängig*
> *KC := R.rechts überlagert KC*
> *Führe semantische Aktionen zu den Bedingungen aus R.rechts aus*
> *CS := KC überlagert CS*

Dieser Algorithmus wird zur Behandlung von Eingaben und Abfragen noch erweitert; siehe hierzu [Olsen]. Darüberhinaus werden Verfahren zur Optimierung beschrieben; zum einen die Eliminierung von Regeln, die in keinem Fall aktivierbar sind, zum anderen (in [Olsen 1992]) eine Repräsentierung von Vektoren und Feldern als Bitvektoren.

Propositional Production Systems sind, wie bereits erwähnt, eine Obermenge von zustandsbasierten Automaten und von *Event–Response–Systemen*, wie [Hill 1986] sie vorschlägt (siehe den vorangegangenen Abschnitt):

- Ein Automat ist ein PPS mit
  - einem Zustandsfeld (`Field`) mit einer Bedingung für jeden Zustand,
  - einem Eingabefeld (`Input Field`) mit einer Bedingung für jede logische Eingabe,
  - einem semantischen Feld (`Semantic Field`) mit einer Bedingung für jede semantische Aktion.
- Ein *Event–Response–System* ist ein PPS mit genau zwei Bedingungen für jedes Feld.

Eine Spezifikation in ERL kann also in einfacher Weise in ein *Propositional Production System* transformiert werden.

3.3 User Interface Design Environment System (UIDE)

Mit der Entwicklung von UIDE ([Foley et al. 1989], [Gieskens, Foley 1991]) wurde Ende der achtziger Jahre begonnen; das System wird heute noch ständig weiterentwickelt (siehe z.B. [Sukaviriya et al. 1993]). Ein für uns wesentliches Merkmal ist eine wissensbasierte Beschreibung der Benutzungsschnittstelle, bei der Aktionen innerhalb der Oberfläche durch Vor– und Nachbedingungen gesteuert werden. Dies wird auch genutzt, um kontextspezifische Hilfe bereitzustellen ([de Graaff 1992], [de Graaff et al. 1993]).

In der Wissensbasis ist die Information repräsentiert, die zur Beschreibung des Systems notwendig ist. Die Beschreibung orientiert sich dabei zunächst an den einzelnen Objekten der Präsentationsschicht; beschrieben werden Aktionen und Bedingungen, die mit der Aktivierung logischer Oberflächeneinheiten (Knöpfen, Menüelementen etc.) verbunden sind.

Die Wissensbasis ist aus mehreren Modulen für die verschiedenen Kategorien von Informationen aufgebaut. Sie umfaßt folgende Objekte und Daten:

- Hierarchische Objektklassen,
- Eigenschaften der Objekte,
- Aktionen, die mit den Objekten ausgeführt werden können,
- Für die Aktionen notwendige Information,
- Vor– und Nachbedingungen für die Aktionen.

Die Wissensbasis wird interaktiv durch Auswahl aus Menüs und sonstigen Benutzereingaben aufgebaut. Aus den so gewonnenen Daten wird eine Beschreibung der Oberfläche in der *Interface Definition Language* (IDL) generiert; die Benutzerin bzw. der Benutzer benötigt bei der Spezifikation keine Kenntnis über deren Syntax.

Weitere Operationen, die auf der Wissensbasis bzw. mit ihrer Hilfe ausgeführt werden können, sind:

- Überprüfung der Schnittstelle im Hinblick auf Konsistenz und Vollständigkeit
- Überführung der Wissensbasis (und damit der durch sie repräsentierten Schnittstelle) in eine andere, funktional äquivalente, Form, um alternative Entwurfskonzepte zu unterstützen
- Bewertung der Benutzungsgeschwindigkeit
- Bereitstellung der Daten für das *Simple User–Interface Management System* (SUIMS)[1], das die Oberfläche realisiert
- Automatische Generierung von Hilfsseiten zur Laufzeit

An dieser Stelle ist insbesondere die Steuerung von Benutzeraktionen durch Vor– und Nachbedingungen von Interesse (vgl. hierzu insbesondere [Gieskens, Foley 1991]).

Vor– und Nachbedingungen sind Objekten, wie den einzelnen Punkten eines Menüs, Fenstern, Knöpfen oder Verschiebebalken zugeordnet, um über die auf einem Objekt ausführbaren Aktionen zu entscheiden. Dazu sind unterschiedlichen Aktionen unterschiedliche Vorbedingungen zugeordnet. Gültige Prädikate werden in einer globalen Liste (*Current State Blackboard*, CSB) gehalten; diese Liste ändert sich, indem mit einer Aktion assoziierte Nachbedingungen etabliert, d.h. der Liste zugefügt werden. Dabei gilt das Prinzip der Negation durch Nichtbeweisbarkeit (*negation–as–failure*, [Clocksin, Mellish 1987]); vorher gültige, jetzt ungültige Prädikate werden einfach aus der Liste entfernt.

Die Prädikate bestimmen damit nicht nur die Semantik, sondern auch das dynamische Verhalten des Systems. Durch Schreiben und Lesen auf der CSB findet eine „anonyme“[2] Kommunikation zwischen Objekten statt. Auf diese Weise findet auch eine Synchronisierung der Aktionen statt.

Jedes Prädikat besteht aus einem Namen und Argumenten. Argumente können Konstanten oder Variablen sein, für die eine beliebige Konstante eingesetzt werden kann. In Nachbedingungen sind darüberhinaus Funktionen erlaubt. Vorbedingungen sind Boole'sche Ausdrücke, Nachbedingungen eine Liste von Änderungen, die auf der CSB durchgeführt werden sollen. Jedes Objekt der Oberfläche (*widget*) besitzt zwei Mengen von Vorbedingungen, eine, die definiert, ob das Objekt sichtbar ist, und eine, die definiert, ob das Objekt aktivierbar ist. Die Zahl der Nachbedingungen ist vom Typ des Objekts abhängig; bei einigen ist nur eine Aktion möglich (z.B. ein Knopf kann lediglich ausgewählt werden; andere Aktionen sind i.d.R. nicht vorgesehen), bei anderen mehrere.

Zur Illustration folgen einige Beispiele für Aktionen (nach [Gieskens, Foley 1991]). Sie beschreiben die Benutzungsoberfläche eines CD–Spielers.

[1] Unter UIMS wird in diesem Zusammenhang die Komponente zur lexikalischen Verarbeitung verstanden (siehe Abschnitt 2.1.1, insbesondere Abbildung 2.3 auf Seite 10).

[2] Das schreibende bzw. lesende Objekt besitzt keine Information, welches Objekt auf die Information zugreift bzw. sie erzeugt hat.

Beispiel 3.3.1 *Auswahl ausführbarer Aktionen: Anhalten einer CD ist nur sinnvoll, wenn die CD nicht bereits angehalten ist.*

```
Button stop
     pre enable:      not status (CD, STOPPED)
     post select:     status (CD, STOPPED)
```

Beispiel 3.3.2 *Abhängigkeit von Aktionen: wird der Knopf zur Titelsuche betätigt, öffnet sich automatisch ein Dialogfenster.*

```
Button search
     post select:     popup (SEARCH)

Popup window search_track_dialogue
     pre visible:     popup (SEARCH)
```

Beispiel 3.3.3 *Variablen und Funktionen: die Variablen* `curr` *und* `total` *erhalten ihre Werte abhängig von den Prädikaten der CSB. Der Wert von* `curr` *wird als Argument der Funktion* `inc` *übergeben, um die Nummer des aktuellen Titels zu erhöhen.*

```
Button next_track
     pre enable:      current_track (curr) and
                      total_tracks (total) and
                      less_than (curr, total)
     post select:     current_track (inc (curr, 1))
```

Die Vor– und Nachbedingungen werden durch zwei Systemkomponenten verwaltet: dem Objektverwalter und dem Prädikatsverwalter. Diese beiden bilden die Verbindung zwischen der Anwendung (einschließlich des UIMS, das für alle die Benutzungsschnittstelle betreffenden Operationen zuständig ist) und den Benutzungsschnittstellenwerkzeugen. Zugriff auf die CSB hat ausschließlich der Prädikatsverwalter; dies stellt sicher, daß bei Änderungen der CSB die Objekte der Oberfläche entsprechend aktualisiert werden. Die Struktur zeigt Abbildung 3.4. Das Zusammenwirken von Widget–Manager und Prädikatsverwalter ist in Abbildung 3.5 zu sehen ([Gieskens, Foley 1991]).

Kontextabhängige Hilfe

Die in dem System enthaltene Kontextinformation (die jeweils aktuelle Menge von Prädikaten in der CSB), wird genutzt, um kontextsensitive Hilfe zu generieren ([de Graaff 1992], [de Graaff et al. 1993]). Aufgrund der vorhandenen Daten ist es möglich, Antworten auf zwei Fragen zu generieren:

- Warum ist ein bestimmtes Objekt nicht aktivierbar?
- Was ist zu tun, um es aktivierbar zu machen?

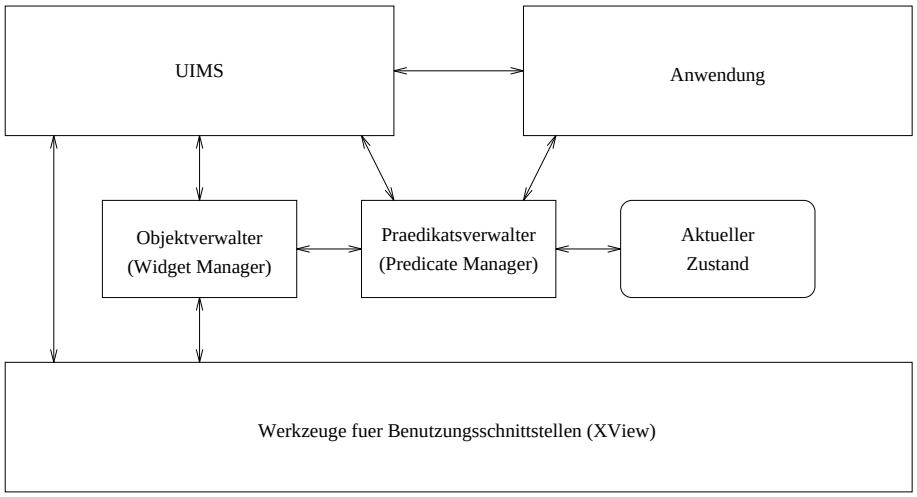

Abbildung 3.4: *Architektur des erweiterten UIDE–Systems*

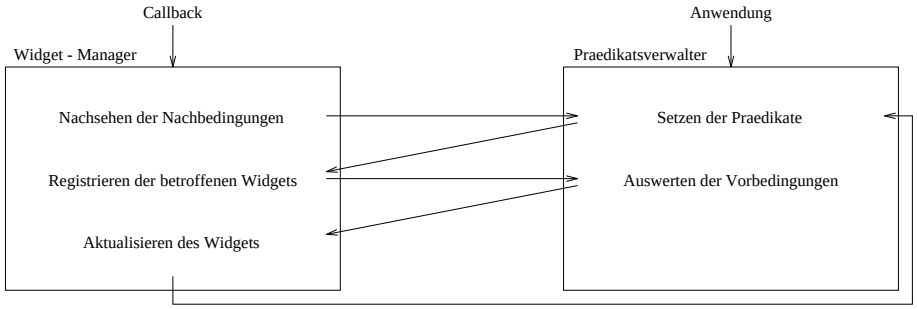

Abbildung 3.5: *Interaktion zwischen Widget–Manager und Prädikatsverwalter*

Die erste Frage kann direkt mit Hilfe der vorhandenen bzw. nicht vorhandenen Prädikate in der CSB beantwortet werden. Um eine Antwort auf die zweite Frage zu finden, werden mit Hilfe eines Planungsalgorithmus die notwendigen Schritte berechnet, die das System in den gewünschten Zustand überführen.

Das System wird ständig weiterentwickelt. So ist in einer neueren Realisierung eine Zuordnung von Aktionen zu zwei Ebenen vorgesehen: einer Anwendungs– und einer Schnittstellenebene. Einer Anwendungsaktion können mehrere Schnittstellenaktionen zugeordnet sein; ein bestimmtes Verhalten der Anwendung kann also durch verschiedene Operationen an der Benutzungsschnittstelle erreicht werden (z.B. Öffnen einer Datei durch Doppelklick mit der Maus oder durch Selektieren und anschließendes Anwählen eines Menüpunkts). Den Schnittstellenaktionen sind wiederum eine oder mehrere Interaktionstechniken angeschlossen, die die konkreten Benutzeraktionen festlegen ([Sukaviriya et al. 1993]).

UIDE dient als Basis für verschiedene weitere Untersuchungen im Bereich von Benutzungsoberflächen. Darauf soll im Rahmen dieser Arbeit nicht mehr eingegangen werden.

3.4 Zusammenfassung

In diesem Kapitel wurden drei Beispiele für Systeme vorgestellt, die die Dialogsteuerung in graphischen Benutzungsschnittstellen mit Hilfe von Regeln realisieren:

- Zunächst SASSAFRAS, bei dem die Dialogsteuerung durch (parametrisierte) Ereignisse angestoßen wird und abhängig von Flags arbeitet. Die Schnittstelle kann in mehrere Module unterteilt werden.
- PPS, das Regeln aufgrund von Zustandsvektoren auswählt, die den internen Status der Steuerung und äußere Information, z.B. Eingabeereignisse repräsentieren.
- UIDE, das das Verhalten einzelner Bildschirmobjekte bei ihrer Aktivierung durch Vor– und Nachbedingungen beschreibt.

Auf die Ansätze werden wir im Abschnitt 4.4 noch einmal zurückkommen. Dort werden sie für einen Vergleich mit unserem Ansatz der situationsorientierten Beschreibung herangezogen.

4

Die Konzeption der Dialogbeschreibung

Wie zuvor bereits dargestellt, soll durch SUSI die Dialogsteuerung mit Hilfe von Konzepten aus der Situationstheorie und der Theorie der Aktionen realisiert werden; d.h. das Zusammenspiel von Benutzungsoberfläche (bzw. Präsentationsebene) und der Anwendung soll mit diesen Konzepten beschrieben werden. Zur Spezifikation der Verbindung von Ereignissen an der Oberfläche und Ereignissen innerhalb der Anwendung wurde eine Beschreibungssprache entwickelt, in der diese durch Operationen (Aktionen) und die Bedingungen für deren Ausführung festgelegt und gemäß der darzustellenden Objekte strukturiert werden kann.

In diesem Kapitel soll nun hauptsächlich auf diese Beschreibungssprache eingegangen werden. Es gliedert sich in fünf Teile:

- Zunächst folgt eine Beschreibung der *semantischen Konzepte* von SUSI. Dies beinhaltet die Semantik von Aktionsbeschreibungen und formale Definitionen der Beschreibungselemente.
- Anschließend wird die *Syntax* der Sprache beschrieben. Dies umfaßt vor allem die Struktur der sprachlichen Beschreibung von Situationstypen und den zugehörigen Situationen des jeweiligen Typs sowie die Syntax der weiteren Beschreibungselemente: Aktionsregeln, Nebenbedingungen und Fakten. Am Schluß dieses Abschnitts wird kurz dargestellt, wie die Schnittstellen zu anderen Komponenten deklariert werden können.
- Danach wird auf die *Architektur* eines möglichen Systems eingegangen. Dabei werden auch anhand eines Beispiels die Schnittstellen und der Informationsfluß erläutert.
- Im vorigen Kapitel wurden verwandte Ansätze zur Dialogbeschreibung dargestellt. Diese werden zu einem *Vergleich* mit SUSI herangezogen.
- Zuletzt soll ein größeres *Beispiel* illustrieren, wie die Dialogsteuerung des situationsorientierten Ansatzes arbeitet.

Die Ergebnisse dieses Kapitels stellen die Grundlage für die im darauffolgenden Kapitel beschriebene Realisierung des SUSI–Systems dar.

4.1 Logische Beschreibung von SUSI

In diesem Abschnitt soll SUSI als logischer Ableitungskalkül beschrieben werden. Dabei wird zunächst von implementierungsbedingten Details abgesehen; auf diese wird in einem späteren Abschnitt eingegangen.

Grundlegene Konzepte von SUSI sind auf der einen Seite *Situationen*, die Ausprägungen bestimmter *Situationstypen* sind. Diese Konzepte sind verwandt mit Konzepten aus der Situationstheorie (siehe Abschnitt 2.2). Auf der anderen Seite enthält SUSI *Aktionen* gemäß der Aktionstheorie, die, von bestimmten Bedingungen abhängig, ausgeführt werden (siehe hierzu ebenfalls Abschnitt 2.2). Sie sind jeweils Situationstypen zugeordnet. Das folgende Beispiel soll illustrieren, wie diese Konzepte zur Modellierung eines Dateimanagers verwendet werden können:

Beispiel 4.1.1 *Bei Dateimanagern, wie sie heute gebräuchlich sind, werden Verzeichnisse in Fenstern dargestellt; die darin enthaltenen Dateien (und Unterverzeichnisse) durch Sinnbilder (Icons). In den einzelnen (Verzeichnis–) Fenstern sind immer gleichartige Operationen möglich (z.B. Dateien kopieren, löschen etc.). Somit können alle diese Fenster demselben Typ zugeordnet werden. Der Inhalt der Verzeichnisse ist aber i.a. unterschiedlich; deswegen ist es sinnvoll, sie durch verschiedene Situationen in* SUSI *zu repräsentieren.*

Weitere Elemente sind *Fakten*, die den internen *Zustand* (den momentanen Kontext) einer Situation repräsentieren; jeder Situation ist eine (möglicherweise leere) Menge von Fakten zugeordnet. Zur Beschreibung komplexer Zusammenhänge dienen *Nebenbedingungen* (Constraints); Formeln der Prädikatenlogik erster Stufe. Sie sind für alle Ausprägungen (Situationen) eines Situationstyps einheitlich festgelegt.

Beispiel 4.1.2 *Betrachten wir wieder den Dateimanager: innerhalb der Beschreibung in* SUSI *kann das Vorhandensein einer Datei in einem Verzeichnis durch ein entsprechendes Fakt in der Faktenmenge der entsprechenden Situation modelliert werden. Eine Nebenbedingung kann z.B. festlegen, unter welchen Bedingungen ein Menüpunkt in einem Pull–Down–Menü anwählbar ist; diese Bedingungen sind in allen Verzeichnisfenstern die gleichen.*

Es soll nun zunächst eine formale Definition der soeben informell eingeführten Konzepte gegeben werden.

4.1.1 Formale Definition der Syntax von SUSI

Grundlegenes Element von SUSI ist die *atomare Formel* bzw. *Relation*:

Definition 4.1.1 (Atomare Formel/Relation) *Eine atomare Formel (Relation) ist ein Term der Form* $t(x_1, \ldots, x_n), n \geq 0$. *Dabei sind die* x_i $(1 \leq i \leq n)$ *logische Konstanten oder logische Variablen.*

Ein *Literal* ist eine nichtnegierte oder negierte atomare Formel:

Definition 4.1.2 (Literal)

1. *Ist A eine atomare Formel, so ist A ein Literal.*
2. *Ist A eine atomare Formel, so ist $\neg A$ ein Literal.*

Mit diesen Definitionen können wir nun zur Darstellung der Elemente einer SUSI–Beschreibung kommen. Zunächst die *Fakten*:

Definition 4.1.3 (Fakt) *Ein Fakt ist eine atomare Formel.*[1]

Als nächstes folgt die Definition von *Nebenbedingungen*. Wie bereits erwähnt, handelt es sich dabei im Prinzip um prädikatenlogische Formeln erster Stufe.

Definition 4.1.4 (Nebenbedingungen (Constraints))

1. *Eine atomare Formel ist eine Nebenbedingung.*
2. *Ist A eine Nebenbedingung, so ist auch $\neg A$ eine Nebenbedingung.*
3. *Sind A und B Nebenbedingungen, so ist auch $A \vee B$ eine Nebenbedingung.*
4. *Sind A und B Nebenbedingungen, so ist auch $A \wedge B$ eine Nebenbedingung.*
5. *Sind A und B Nebenbedingungen, so ist auch $A \rightarrow B$ eine Nebenbedingung.*

Obwohl atomare Formeln gemäß dieser Definition Nebenbedingungen sind, werden sie in der Praxis nicht als solche auftauchen.

Das letzte Element der Beschreibungssprache von SUSI ist die *Aktionsregel*. Die Menge der Aktionsregeln beschreibt das tatsächliche Verhalten einer Situation zur Laufzeit. Zuvor müssen wir klären, was wir unter *Vor–* bzw. *Nachbedingungen* verstehen:

Definition 4.1.5 (Vorbedingung)

1. *Ein Literal ist eine Vorbedingung.*
2. *Sind A und B Vorbedingungen, so ist auch $A \wedge B$ eine Vorbedingung.*

In gleicher Weise werden Nachbedingungen definiert:

Definition 4.1.6 (Nachbedingung)

1. *Ein Literal ist eine Nachbedingung.*
2. *Sind A und B Nachbedingungen, so ist auch $A \wedge B$ eine Nachbedingung.*

Einzelne Literale können dabei unterschiedliche Bedeutung haben; davon soll zunächst abgesehen werden.

Definition 4.1.7 (Aktionsregel) *Eine Aktionsregel ist ein 3–Tupel (B, V, N). Dabei ist B ein (eindeutiger) Bezeichner, V eine Vorbedingung und N eine Nachbedingung.*

[1] Da wir in der vorliegenden Implementierung Bereichsbeschränktheit fordern, können als Argumente nur Konstanten auftauchen.

Fakten, Nebenbedingungen und Aktionsregeln bilden die Grundbausteine für Situationstypen und Situationen. Diese sind wie folgt definiert:

Definition 4.1.8 (Situationstyp) *Ein Situationstyp ist ein 3–Tupel (B, N, A). Dabei ist B ein (eindeutiger) Bezeichner, N eine Menge von Nebenbedingungen und A eine Menge von Aktionsregeln.*

Definition 4.1.9 (Situation) *Eine Situation ist ein 3–Tupel (B, F, S). Dabei ist B ein (eindeutiger) Bezeichner, F eine Menge von Fakten und S ein Situationstyp.*

Die Auswahl der anzuwendenden Aktionsregeln ist abhängig von einem Trigger:

Definition 4.1.10 (Trigger) *Als Trigger bezeichnen wir eine besonders ausgezeichnete Relation in der Vorbedingung einer Aktionsregel oder eine global vorhandene und durch Angabe von (Ziel–) Situation und Situationstyp qualifizierte Relation. Trigger beeinflussen die Auswahl anwendbarer (s.u.) Aktionsregeln, indem für die Anwendbarkeit der Regel der globale Trigger mit dem in der Aktionsregel enthaltenen Trigger unifizierbar sein muß.*

4.1.2 Ableitung in SUSI

Die Ableitung in SUSI kann auf hat zwei Ebenen betrachtet werden: auf der prozeduralen Ebene der Aktionen und auf der logischen Ebene der Bedingungen für deren Ausführung.

- Auf der höheren Ebene, auf der die prozedurale Seite von SUSI festgelegt wird, finden Ableitungen durch das Ausführen („feuern“) von Aktionsregeln statt, die nach Maßgabe des aktuellen Triggers ausgewählt werden. Dabei werden, abhängig von der Gültigkeit der Vorbedingung, neue, in der Nachbedingung enthaltene positive Fakten etabliert und negative gelöscht. Gleichzeitig können neue Trigger abgesetzt werden, von denen dann die Auswahl der nächsten zu aktivierenden Aktionsregel abhängt.
- Auf der niedrigeren Ebene findet eine logische Ableitung statt, um die Vorbedingungen von Aktionsregeln zu testen. Prinzipiell ist dies eine Ableitung in einem Kalkül der Prädikatenlogik erster Stufe. Aus Gründen der Effizienz und leichteren Handhabbarkeit wird man bei der Realisierung Einschränkungen in Kauf nehmen müssen.

Im folgenden soll nun das semantische Konzept von SUSI unter diesen beiden Aspekten betrachtet werden.

Ableitung durch Aktionsregeln

Nach dem eben gesagten hängt die Ausführung einer Aktionsregel von der Gültigkeit (Ableitbarkeit, s.u.) der Vorbedingung ab. Die Vorbedingung ist eine Konjunktion von Literalen, welche sich in vier Kategorien einteilen lassen:

- Am Beginn der Vorbedingung jeder Aktionsregel befindet sich der Trigger als ausgezeichnete Relation. Bei jeder Aktionsregel wird versucht, diese Triggerrelation mit einem globalen Trigger (der z.B. aus einer Warteschlange entnommen wird) zu unifizieren. Gelingt dies nicht, so ist die Aktionsregel (in diesem Moment) nicht anwendbar. Sonst wird (unter Berücksichtigung der sich ergebenden Substitutionen) mit der Ableitung der Vorbedingung dieser Regel fortgefahren.

- Die zumeist auftretenden Literale sind positive oder negative Atome, die mit Hilfe der in der aktuellen Situation vorhandenen Information (Fakten und Nebenbedingungen), wie weiter unten beschrieben, abgeleitet werden.

- Bei der Entscheidung über die Ausführung einer Aktion muß mitunter auf in anderen Situationen repräsentierte Information zugegriffen werden. Um dies zu ermöglichen, können Literale auch extern, d.h. innerhalb einer anderen Situation bewiesen werden. Dazu zählen auch die „virtuellen" Situationen, die die Präsentationsschicht und die Anwendung repräsentieren.

- Semantische Aktionen — i.d.R. Aktionen in der Anwendung — werden durch spezielle Literale ausgelöst, denen ein Anwendungsbefehl zugeordnet ist. Hier ergibt sich der Wahrheitswert aus dem erfolgreichen (*wahr*) oder nicht erfolgreichen (*falsch*) Abschluß der Aktion.

Folgendes Beispiel soll den Bezug zu dem bereits erwähnten Dateimanager herstellen:

> Beispiel 4.1.3 *Um eine Datei in ein anderes Verzeichnis zu kopieren, müssen verschiedene Voraussetzungen erfüllt sein: zunächst muß das entsprechende Sinnbild auf der Oberfläche zum anderen Ordner gezogen werden; daraus resultiert ein Trigger. Es muß dann aus der Faktenbasis* der aktuellen Situation *z.B. der Besitzer der Datei ermittelt werden. Aus der Faktenbasis* einer anderen Situation *(der, die das Zielverzeichnis repräsentiert) muß z.B. ermittelt werden, ob bereits eine Datei mit demselben Namen existiert. Sind die entsprechenden (und evtl. noch weitere) Literale der Vorbedingung erfolgreich abgeleitet, so wird die* semantische Aktion (`copy`) *in der Anwendung aktiviert.*

Aus logischer Sicht werden alle Relationen der Vorbedingung bewiesen; d.h. sie werden an einen Auswertungsalgorithmus übergeben, der einen logischen Wert (*wahr* oder *falsch*) zurückliefert. Dies gilt auch für die semantischen Aktionen, die Seiteneffekte erzeugen. Obwohl in SUSI — der Idee nach — die Bedingungen für eine semantische Aktion vor deren Ausführung geprüft werden, können solche Aktionen unerwartet fehlschlagen. In diesem Fall wird das Ergebnis der Ableitung der Wert *falsch* sein.

> Beispiel 4.1.4 *Betrachten wir den bereits beschriebenen Kopiervorgang: zunächst werden die relevanten Bedingungen überprüft, dann wird die semantische Aktion* `copy` *angestoßen. Wird nun durch die Aktivität eines anderen Prozesses, die nicht vorherzusehen war, der verbleibende Platz auf dem Speichermedium zu gering, so wird der Kopiervorgang erfolglos abgebrochen.*

Für solche Fälle kann beispielsweise eine weitere (Fehler–) Aktion vorgesehen sein. Da als Bedingung für deren Ausführung das Fehlschlagen der semantischen Aktion abgeleitet werden muß

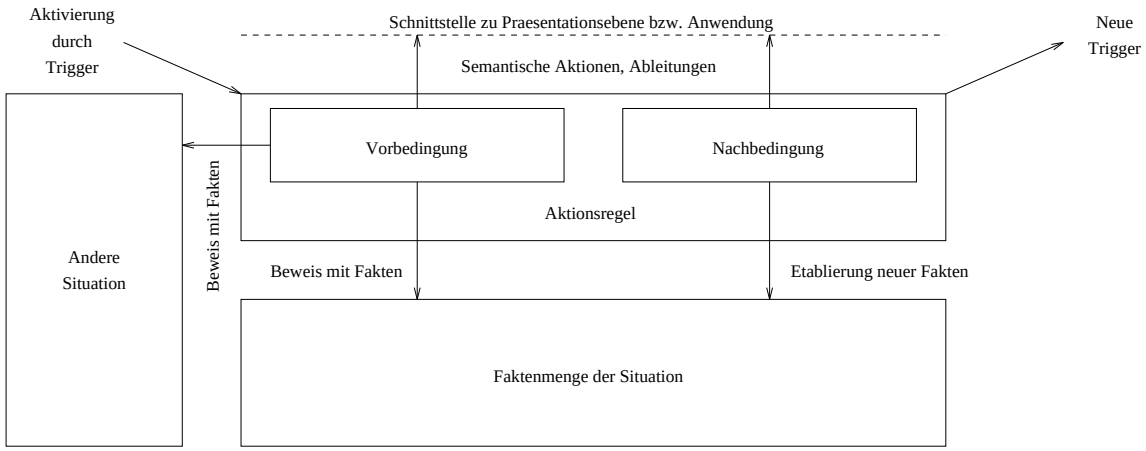

Abbildung 4.1: *Zusammenwirken einer Aktionsregel mit anderen Systemelementen.*

(und es gleichzeitig nicht erwünscht ist, diesen Seiteneffekt zweimal anzustoßen), ist dafür die Information über das Ergebnis der vorherigen Ableitung zwischenzuspeichern.

Ist die Vorbedingung vollständig erfolgreich abgeleitet, erfolgt die Bearbeitung der Nachbedingung. Auch hier gibt es verschiedene Kategorien von Relationen, die unterschiedlich behandelt werden.

- Im Normalfall beschreiben die Relationen der Nachbedingung eine Änderung der Faktenmenge der Situation. Dabei werden positive Literale der Faktenmenge zugeschlagen, mit negativen Literalen unifizierbare Fakten aus der Faktenmenge entfernt.
- Bei Fakten, die Seiteneffekte erzeugen (hier i.d.R. auf die Präsentationsebene), werden zusätzlich Operationen (etwa durch Funktionsaufrufe), getrennt für den negativen und den positiven Fall, angestoßen.
- Um weitere Aktionsregeln zu aktivieren, können Nachbedingungen auch Trigger enthalten, die in die entsprechende Warteschlange aufgenommen werden.

Abbildung 4.1 soll das Zusammenwirken der Aktionsregeln mit anderen Elementen noch einmal verdeutlichen.

Nach dieser informellen Beschreibung soll nun eine formale Beschreibung von einigen Aspekten der Aktionsregeln vorgenommen werden.

Wie bereits gesehen, werden Aktionsregeln abhängig von einer besonderen Relation aktiviert, dem Trigger. Mit seiner Hilfe wird die nächste auszuführende Aktion ausgewählt. Wir können somit zunächst die Anwendbarkeit einer Aktionsregel bei gegebenem Trigger genauer fassen:

> Definition 4.1.11 (Anwendbarkeit) *Sei gegeben eine Menge von Aktionsregeln AR und ein Trigger T. Eine Regel $R \in AR$ heißt* anwendbar für T, *wenn T und die in der Vorbedingung von R enthaltene Trigger–Relation unifizierbar sind.*

Es ist dabei möglich, daß für einen Trigger keine anwendbare Aktionsregel existiert.

Die Anwendbarkeit ist lediglich eine notwendige Bedingung für die *Ausführbarkeit*. Diese schließt die *Ableitbarkeit* der gesamten Vorbedingung ein.

> Definition 4.1.12 (Ausführbarkeit) *Eine Aktionsregel R ist bei gegebenem Trigger T* ausführbar*, wenn sie*
>
> 1. *anwendbar und*
> 2. *die Vorbedingung ableitbar ist.*

> Definition 4.1.13 (Ableitbarkeit der Vorbedingung) *Die Vorbedingung einer Aktionsregel ist (theoretisch) ableitbar, wenn sie im Sinne der semantischen Ableitbarkeit der Prädikatenlogik erster Stufe wahr ist. Formal: Sei F die Menge der Fakten einer Situation, N die Menge der Nebenbedingungen des entsprechenden Situationstyps. Dann ist die Vorbedingung V einer Aktionsregel ableitbar, wenn gilt: $F \cup N \models V$.*

In der Praxis wird die semantische Gültigkeit durch die syntaktische Herleitbarkeit innerhalb eines geeigneten Kalküls, beispielsweise des Resolutionskalküls, ersetzt.

Wird eine ausführbare Aktionsregel dann tatsächlich ausgeführt, so kann sie in dreierlei Weise wirken:

- Die Faktenmenge der aktuellen Situation kann in der bereits beschriebenen Weise verändert werden.
- Es können neue Trigger generiert werden, die in eine Warteschlange (bzw. einen Keller) aufgenommen werden.
- Entsprechend der Änderung der Faktenmenge können Auswirkungen auf die Präsentationsebene durch die in der Dialogspezifikation definierten Seiteneffekte auftreten.

Existieren für den aktuellen Trigger mehrere ausführbare Aktionsregeln, so muß eine Entscheidung getroffen werden, mit welcher Regel fortgefahren werden soll. Dafür gibt es zwei Möglichkeiten:

- Es kann eine *Auswahlfunktion* vorgesehen werden. Beispielsweise kann die in textueller Reihenfolge erste Aktionsregel selektiert werden.
- Es können — nacheinander oder in parallelen Prozessen — *alle* anwendbaren Regeln aktiviert werden.

Am wenigsten Probleme scheint dabei die erste Variante zu machen; dieser Weg ist auch z.B. bei der logischen Programmiersprache PROLOG (siehe etwa [Clocksin, Mellish 1987]) beschritten worden. Im zweiten Fall — insbesondere bei paralleler Ausführung — können Probleme auftreten. Werden z.B. zwei Regeln parallel bearbeitet, die sich in widersprüchlicher Weise auf die Faktenbasis der Situation auswirken, ist nicht vorherzusagen, welche Form diese anschließend hat, da nicht eindeutig bestimmt ist, welcher Prozeß zuerst beendet ist. Deswegen bietet sich die Auswahl einer Regel zur Ausführung an. Erweist sich diese als nicht ausführbar, wird zur nächsten anwendbaren Regel übergegangen.

Logische Ableitung durch Fakten und Nebenbedingungen

Wir haben oben gesehen, daß die Ausführung einer Aktionsregel — neben dem Trigger — von der Ableitung ihrer Vorbedingung abhängt. Diese Ableitung erfolgt prinzipiell im Rahmen der Prädikatenlogik erster Stufe mit Hilfe eines Kalküls. Ableitbarkeit in einem Kalkül ist allgemein folgendermaßen definiert ([Menzel, Schmitt 1993]):

Definition 4.1.14 (Syntaktische Ableitung) *Gegeben sei eine Sprache (Δ, L) mit Δ ein Alphabet, L eine entscheidbare Teilmenge von Δ^*. Kal sei ein Kalkül, M eine Teilmenge von L. Eine* Ableitung *aus M in Kal ist eine Folge*

$$(u_1, \ldots, u_m)$$

von Wörtern in L, so daß für jedes $i \in \{1, \ldots, m\}$ gilt:

- *u_i ist Axiom, oder*
- *$u_i \in M$, oder*
- *es gibt eine Regel $R \in Kal$ einer Stelligkeit $n \geq 1$ sowie Indizes $j_1, \ldots, j_n \in \{1, \ldots, i-1\}$, so daß $(u_{j_1}, \ldots, u_{j_n}, u_i) \in R$.*

$u \in L$ heißt ableitbar *aus M in Kal, kurz*

$$M \vdash_{Kal} u$$

genau dann, wenn es eine Ableitung $(u_1, \ldots, u_m)$ in Kal gibt mit $u_m = u$.

Im folgenden soll nun das Vorgehen bei der logischen Ableitung von Bedingungen in SUSI erläutert werden.

Der Algorithmus zum Beweis von Relationen (das sind i.d.R. die Vorbedingungen einer Aktionsregel) lehnt sich an das *Recursive Query/Recursive Subquery*–Verfahren an, das in [Bancilhon, Ramakrishnan 1988] beschrieben wird. Es handelt sich dabei um ein rekursives Verfahren, das dem Ableitungsmechanismus von PROLOG ähnlich ist. Im Unterschied dazu leitet es jedoch nicht nur jeweils eine Lösung her, sondern erzeugt alle möglichen Substitutionen in einem Durchlauf.

Auf diese Weise werden alle Lösungen unmittelbar erzeugt; das sonst übliche Zurücksetzen (*Backtracking*) ist nicht erforderlich. Der verwendete Algorithmus hat die folgende Form:

Algorithmus 4.1.1 (Ableitung von Relationen)

evaluate (*Anfrage, Situation, Anfragestapel*)

(Auswahl der anzuwendenden Regeln *)*

if *Anfrage seit letzter Änderung der Faktenmenge abgeleitet*
then *Wähle anwendbare Regeln aus Menge bereits abgeleiteter Relationen.*
else *Wähle anwendbare Regeln aus Menge der Fakten und Nebenbedingungen.*
end

(Ableitung mit Hilfe der ausgewählten Regeln *)*

while *noch anwendbare Regeln vorhanden*
begin
Unifiziere Kopf der Regel mit Anfrage, füge notwendige Substitutionen zu Menge hinzu.
while *noch Relationen im Rumpf nicht abgeleitet* **and** *alle Ableitungen erfolgreich*
begin
Wähle nächste Relation als $Anfrage_{neu}$
evaluate (*Anfrage*$_{neu}$, *Situation, cons (Anfrage*$_{neu}$, *Anfragestapel)*)
Schneide errechnete Substitutionsmenge mit bisheriger Menge.
end
Vereinige neu errechnete Substitutionsmenge mit bisheriger Menge.
end
end

Die eigentliche Ableitung findet dabei in der **while**–Schleife statt. Die Abfrage zu Beginn entscheidet über die anzuwendenden Regeln. Wurde die abzuleitende Relation nach der letzten Ausführung einer Regel bereits abgeleitet, so ist eine erneute Ableitung nicht erforderlich; die bereits berechnete Information kann genutzt werden.

Ein Beispiel zeigt die Bildung der Substitutionsmengen während der Ableitung.

Beispiel 4.1.5

Fakten:	$a(eins)$ $a(zwei)$
Anfrage:	$a(X)$
Substitutionen:	$\{(X \leftarrow eins), (X \leftarrow zwei)\}$
Ergebnis:	$a(eins), a(zwei)$

Während der Ableitung werden keine Variablen propagiert; die Instantiierungen finden erst am Ende statt. Eine Anfrage kann auf zwei Arten fehlschlagen:

- Es findet sich keine Regel, deren Kopf mit der Anfrage unifizierbar ist.
- Die Anfrage ist ableitbar; die dabei errechnete Substitution ist aber nicht mit den bereits vorhandenen Substitutionen verträglich.

Was unter einer verträglichen Substitution verstanden werden soll, ist Inhalt der folgenden Definition:

Definition 4.1.15 (Verträgliche Substitutionen) *Eine Substitution ist mit einer Menge von Substitutionen* verträglich*, wenn eine der folgenden Bedingungen gilt:*

- *In der Substitutionsmenge ist noch keine Substitution für die zu ersetzende Variable vorhanden. In diesem Fall wird die neue Substitution der Menge hinzugefügt.*

- *In der Substitutionsmenge ist bereits eine entsprechende Ersetzung enthalten. Dann muß die Schnittmenge der neuen und der alten Substitution nichtleer sein, d.h. es muß eine Substitution existieren, die den Variablen, denen bereits Werte zugewiesen sind, die gleichen Werte zuweist. Alle anderen Substitutionen werden aus der Menge entfernt.*

Substitutionen sind also verträglich, wenn sie eine *nichtleere Schnittmenge* besitzen oder wenn *genau eine* der zu schneidenden Mengen nichtleer ist.

Dies illustrieren die folgenden beiden Beispiele. Im ersten haben die Mengen der möglichen Substitutionen eine nichtleere Schnittmenge; die Ableitung ist erfolgreich. Im zweiten Beispiel ist die Schnittmenge leer; die Ableitung schlägt somit fehl.

Beispiel 4.1.6

Fakten:	$a(falsch)$ $a(richtig)$ $b(richtig)$
Nebenbedingung:	$a(X)$ **and** $b(X)$ **impl** $c(X)$
Anfrage:	$c(X)$
Ableitung von $a(X)$*:*	$a(\{falsch, richtig\})$
Substitution:	$\{(X \leftarrow falsch), (X \leftarrow richtig)\}$
Ableitung von $b(X)$*:*	$b(\{richtig\})$
Substitution:	$\{(X \leftarrow richtig)\}$
Ergebnis:	$c(\{richtig\})$
wegen	$\{(X \leftarrow falsch), (X \leftarrow richtig)\}$ $\cap\{(X \leftarrow richtig)\}$ $= \{(X \leftarrow richtig)\}$

Beispiel 4.1.7

Fakten:	$a(falsch)$ $b(auchFalsch)$
Nebenbedingung:	$a(X)$ **and** $b(X)$ **impl** $c(X)$
Anfrage:	$c(X)$
Ableitung von $a(X)$*:*	$a(\{falsch\})$
Substitution:	$\{(X \leftarrow falsch)\}$
Ableitung von $b(X)$*:*	$b(\{auchFalsch\})$
Substitution:	$\{(X \leftarrow auchFalsch)\}$
Ergebnis:	$c(\{\})$, *also nicht ableitbar*
wegen	$\{(X \leftarrow falsch)\} \cap \{(X \leftarrow auchFalsch)\} = \{\}$

Gibt es mehrere Regeln, durch die die Anfrage bewiesen werden kann, so werden alle Ableitungen durchgeführt und die entstehenden Substitutionen miteinander vereinigt. Wird eine negierte

Formel abgeleitet, so werden die bei der Ableitung gefundenen Substitutionen aus der Menge entfernt.

Das Vorgehen kann formal folgendermaßen beschrieben werden: eine Menge von Substitutionen zu einer Anfrage $P(\overline{x})$ an eine Datenbasis DB kann als eine Menge $\mathcal{M}(P(\overline{x}))$ von Lösungsvektoren aufgefaßt werden:

$$\mathcal{M}(P(\overline{x})) = \{\overline{x} | DB \vdash P(\overline{x})\}$$

Dabei gilt für die Gesamtheit der Regeln $c_1 \leftarrow a_1, c_2 \leftarrow a_2, \ldots, c_n \leftarrow a_n$ $(n \geq 1)$, deren Köpfe c_i $(1 \leq i \leq n)$ mit $P(\overline{x})$ unifizierbar sind

$$\mathcal{M}(P(\overline{x})) = \mathcal{M}(a_1) \cup \mathcal{M}(a_2) \cup \ldots \cup \mathcal{M}(a_n)$$

und für jedes Fakt $F(\overline{y_i})$ $(1 \leq i \leq n)$ entsprechend

$$\mathcal{M}(P(\overline{x})) = \overline{y_1} \cup \ldots \cup \overline{y_n}$$

Für zusammengesetzte Formeln gilt:

$$\begin{aligned} \mathcal{M}(P(\overline{x}) \wedge Q(\overline{x})) &= \mathcal{M}(P(\overline{x})) \cap \mathcal{M}(Q(\overline{x})) \\ \mathcal{M}(P(\overline{x}) \vee Q(\overline{x})) &= \mathcal{M}(P(\overline{x})) \cup \mathcal{M}(Q(\overline{x})) \\ \mathcal{M}(P(\overline{x}) \wedge \neg Q(\overline{x})) &= \mathcal{M}(P(\overline{x})) \setminus \mathcal{M}(Q(\overline{x})) \end{aligned}$$

Es ist zu beachten, daß zur Ableitung negativer Literale die Argumente bereits instantiiert sein müssen, d.h., wenn es sich nicht ohnehin um Konstanten handelt, bereits eine Substitution vorliegen muß (siehe dazu [Schmitt 1991]).

Satz 4.1.1 *Ist eine Anfrage $P(\overline{x})$ mit allen g_i und c_i der Regelmenge*

$$\begin{array}{lll} g_1 & \longleftarrow & a_{11} \wedge \ldots \wedge a_{1n_1} \wedge \neg b_{11} \wedge \ldots \wedge \neg b_{1m} \vee \\ & \ldots & \\ g_l & \longleftarrow & a_{l1} \wedge \ldots \wedge a_{ln_l} \wedge \neg b_{l1} \wedge \ldots \wedge \neg b_{lm_l} \vee \\ c_1 & \vee & \\ & \ldots & \\ c_k & & \end{array}$$

unifizierbar, so ist die Menge $\mathcal{M}(P(\overline{x}))$ der möglichen Lösungsvektoren $\overline{x}$ zur Anfrage $P(\overline{x})$

$$\begin{array}{l} \mathcal{M}a_{11} \cap \ldots \cap \mathcal{M}a_{1n_1} \setminus (\mathcal{M}(b_{11}) \cup \ldots \cup \mathcal{M}(b_{1m_1})) \cup \\ \ldots \\ \mathcal{M}a_{l1} \cap \ldots \cap \mathcal{M}a_{ln_l} \setminus (\mathcal{M}(b_{l1}) \cup \ldots \cup \mathcal{M}(b_{lm_l})) \cup \\ \mathcal{M}(c_1) \cup \ldots \cup \mathcal{M}(c_k) \end{array}$$

Beweis: Nach den oben gemachten Aussagen gilt, da $P(\overline{x})$ mit allen g_i und c_i unifizierbar ist:

$$\begin{aligned}
\mathcal{M}(P(\overline{x})) &= \mathcal{M}(a_{11} \wedge \ldots \wedge a_{1n_1} \wedge \neg b_{11} \wedge \ldots \wedge \neg b_{1m_1}) \cup \\
&\quad \ldots \\
&\quad \mathcal{M}(a_{l1} \wedge \ldots \wedge a_{ln_j} \wedge \neg b_{l1} \wedge \ldots \wedge \neg b_{lm_j}) \cup \\
&\quad \mathcal{M}(c_1) \cup \ldots \cup \mathcal{M}(c_k) \\
&= \mathcal{M}(a_{11} \wedge \ldots \wedge a_{1n_1}) \setminus \mathcal{M}(b_{11}) \setminus \ldots \setminus \mathcal{M}(b_{1m_1}) \cup \\
&\quad \ldots \\
&\quad \mathcal{M}(a_{l1} \wedge \ldots \wedge a_{ln_l}) \setminus \mathcal{M}(b_{l1}) \setminus \ldots \setminus \mathcal{M}(b_{lm_l}) \cup \\
&\quad \mathcal{M}(c_1) \cup \ldots \cup \mathcal{M}(c_k) \\
&= \mathcal{M}(a_{11}) \cap \ldots \cap \mathcal{M}(a_{1n_1}) \setminus (\mathcal{M}(b_{11}) \cup \ldots \cup \mathcal{M}(b_{1m_1})) \cup \\
&\quad \ldots \\
&\quad \mathcal{M}(a_{l1}) \cap \ldots \cap \mathcal{M}(a_{ln_l}) \setminus (\mathcal{M}(b_{l1}) \cup \ldots \cup \mathcal{M}(b_{lm_l})) \cup \\
&\quad \mathcal{M}(c_1) \cup \ldots \cup \mathcal{M}(c_k)
\end{aligned}$$

■

Bei der Ableitung können darüberhinaus die bereits abgeleiteten Relationen in einer Liste gesammelt werden. Tritt die gleiche Anfrage ein zweites Mal auf, so wird auf diese bereits errechneten Ergebnisse zurückgegriffen. Dies hat zwei Vorzüge:

- Erhöhte Effizienz: da gleiche Berechnungen nicht mehrfach durchgeführt werden, wird die Ableitung beschleunigt.
- Werden Relationen ausgewertet, die Seiteneffekte erzeugen (auf die Anwendung oder die Benutzungsschnittstelle), kann eine Fehlerbehandlung erfolgen. Wird hier die gleiche Relation ein zweites Mal ausgewertet, so tritt der Seiteneffekt nicht mehr auf.

Dabei ist aber zu beachten: werden die Vorbedingungen einer Aktionsregel alle erfolgreich abgeleitet, so findet eine Modifikation der Faktenmenge statt. Dabei können Inkonsistenzen zwischen Fakten (und Nebenbedingungen) sowie den zuletzt abgeleiteten Relationen entstehen. Deswegen müssen diese bei erfolgreicher Ausführung einer Aktionsregel gelöscht werden. Sind die Vorbedingungen einer Aktionsregel nicht beweisbar, ist das Löschen, da keine Änderung der Faktenmenge eintritt, nicht erforderlich.

4.1.3 Nicht–Horn–Klauseln

Bei der Benutzung des Systems wird die Möglichkeit angestrebt, Nebenbedingungen nicht nur als Hornklauseln, sondern als allgemeine prädikatenlogische Formeln anzugeben. Dies würde es insbesondere ermöglichen, nachprüfbare Integritätsbedingungen für die Datenbasis zur Verfügung zu haben, die bei der Beschränkung auf Hornklauselmengen (die stets ein Modell besitzen; dies wird in [Schmitt 1991] nachgewiesen) nicht vorhanden sind. Solche Nebenbedingungen werden nach dem Einlesen in Klauseln umgeformt. Da bei den dabei entstehenden

Klauseln, die die Horn–Eigenschaft nicht besitzen, im Gegensatz zu den allgemeinen Hornklauseln in Prolog kein Kopf der Klausel mehr vorgegeben ist, ist ein Verfahren notwendig, diesen Kopf, nach dem dann die Ableitung erfolgen kann, zu bestimmen.

Yahya und Henschen schlagen dafür das folgende Verfahren vor ([Yahya, Henschen 1985]):

- Ist unter den positiven Literalen der Klausel *genau eine*, die nicht durch andere Klauseln ableitbar ist, so wird diese zum Kopf der Klausel; die anderen werden in den Rumpf übernommen.
- Sind die positiven Klauseln alle anderweitig ableitbar, so dient die Klausel als Integritätsbedingung; d.h. wenn die linke Seite der Implikation gilt, muß auch die rechte gelten. Es ist immer dann sinnvoll, dies zu überprüfen, wenn sich durch eine Aktionsregel die Faktenbasis ändert.
- Problematisch wird es, wenn mehr als ein nicht anderweitig ableitbares positives Literal existiert. Nach Yahya und Henschen ist dann eine Ableitung nicht mehr möglich. Gegebenenfalls muß diese Klausel mit Fehlermeldung zurückgewiesen werden.

Ein weiteres Problem sind mögliche Zyklen in der Ableitungsfolge, die über mehrere Klauseln gehen. Beispiel 4.1.8 zeigt einen solchen Zyklus.

Beispiel 4.1.8 *Zwei Regeln, bei deren Ableitung ein Zyklus entstehen kann*

P (x, y) **and** *SIB (y, z)* **impl** *UN (x, z)* **or** *AUN (x, z)*
NIE (t, u) **impl** *UN (u, t)* **or** *AUN (u, t)*

Durch solche Zyklen können sich Verklemmungen ergeben.

Von Yahya und Henschen wird dazu vorgeschlagen, einen Graphen folgendermaßen zu konstruieren: Seien $c_i, i \in \{1, \ldots, n\}$ die (positiven) Klauseln, die durch entfernen aller negativen Literale aus den ursprünglichen Klauseln erhalten werden, l_i ein ausgezeichntetes Literal, für das die Klausel als Ableitungsregel dienen soll. Wir erhalten:

$$\begin{aligned} c_1 &= m_{11} \vee \ldots \vee m_{1j_1} \vee l_1 \\ &\ldots \\ c_n &= m_{n1} \vee \ldots \vee m_{nj_n} \vee l_n \end{aligned}$$

Daraus wird ein gerichteter Graph konstruiert mit Knoten der Form

m_{i1}	m_{i2}	$\ldots$	m_{ij_i}	l_i

und Verbindungen zwischen den m_{ij} und den l_k genau dann, wenn sie unifizierbar sind. Dann gilt folgender Satz:

Satz 4.1.2 *Ist der nach der oben beschriebenen Methode aus den Ableitungsregeln der Datenbasis konstruierte Graph* azyklisch, *so ergeben sich bei der Ableitung keine Probleme aus der fehlenden Horn–Eigenschaft der Regeln. Unter der* Closed World Assumption *(CWA) können komplexe Anfragen ausgewertet werden, indem die Ergebnisse der Auswertung der Atome durch Mengenoperationen (s.o.) verknüpft werden.*

Beweis: Siehe [Yahya, Henschen 1985]. ■

Yahya und Henschen weisen abschließend darauf hin, daß — aufgrund der wesentlich höheren Komplexität der Ableitung — nach Möglichkeit auf Datenbasen zurückgegriffen werden sollte, die sich auf Hornklauseln beschränken ([Yahya, Henschen 1985]). Gerade im Zusammenhang mit zeitkritischen Systemen, wie es Benutzungsoberflächen sind, ist genau zu überlegen, ob der Preis einer erhöhten Ausführungszeit für die erhöhte Ausdrucksmächtigkeit gezahlt werden kann.

4.2 Die Grammatik der Sprache SUSI

Wie im vorherigen Abschnitt dargestellt, werden durch SUSI zunächst *Situationen* beschrieben. Diese haben einen internen Zustand, der durch eine Menge von *Fakten* festgelegt ist. Diese Menge verändert sich zur Laufzeit; es werden in der Spezifikation die im Moment des Systemstarts gültigen Fakten festgelegt.

Die Situationen haben jeweils einen bestimmten *Situationstyp*; dieser stellt somit die übergeordnete Struktur dar. Ihm sind (neben den Aktionsregeln und den Nebenbedingungen) alle Situationen seines Typs zugeordnet. Neben den unmittelbar einer Situation zugeordneten Fakten kann zu jedem Situationstyp auch eine Menge von Fakten angegeben werden, mit denen jede Situation des Typs zunächst ausgestattet wird. Darüberhinaus wird jedem Situationstyp ein (eindeutiger) Name zugeordnet.

In ihrer Grobstruktur sieht damit eine SUSI–Spezifikation aus wie aus Abbildung 4.2 ersichtlich. Sie besteht aus drei Abschnitten die mit den Schlüsselwörtern `actionrules`, `constraints` und `facts` überschrieben sind und die Mengen der jeweiligen Konstrukte enthalten.

Dabei sind auf oberer syntaktischer Ebene die einzelnen Situationen noch nicht spezifiziert. Dies hat seinen Grund darin, daß Situationen des gleichen Typs sich (abgesehen von ihrem eindeutigen Bezeichner) nur durch ihre Faktenmenge unterscheiden. Definiert werden die zu Beginn vorhandenen Situationen deswegen im Abschnitt `facts`; beim Einlesen dieses Abschnitts muß dann ein entsprechendes Objekt für die spezifizierten Situationen mit der zugehörigen Faktenmenge (einschließlich der globalen Fakten des Situationstyps) angelegt werden. Entstehen zur Laufzeit neue Situationen (beispielsweise durch Öffnen eines neuen Fensters), so werden entsprechende Objekte erzeugt.

Der Abschnitt *facts* hat damit letztendlich die in der Abbildung 4.3 gezeigte Form.

Aktionsregeln

Aktionsregeln setzen sich aus einer *Vorbedingung* und einer *Nachbedingung* zusammen, die in der Beschreibung durch einen Pfeil („`-->`“) getrennt sind:

Vorbedingung `-->` *Nachbedingung*

Die Vorbedingung und die Nachbedingung sind beides Konjunktionen von Relationen. Die Konjunktion wird durch das Schlüsselwort „`and`“ ausgedrückt, zur Negation wird ein „`not`“ vorangestellt und die Relation geklammert. Argumente von Relationen können Konstanten oder Variablen sein; zu ihrer Unterscheidung wird gemäß der Prolog–Konvention verfahren, d.h. Variablen beginnen mit Großbuchstaben oder Unterstrich, alles andere sind Konstanten (siehe [Clocksin, Mellish 1987]).

```
situationtypes

    Name des ersten Situationstyps

        actionrules
        Aktionsregeln
        end

        constraints
        Constraints (Nebenbedingungen)
        end

        facts
        Globale Fakten des Situationstyps und Fakten der Situationen
        end

    end

    Name des zweiten Situationstyps
        ...
    end

    ...
end
```

Abbildung 4.2: *Die syntaktische Grobstruktur einer* SUSI*–Beschreibung*

Spezielle Relationen sind zum einen diejenigen, die in anderen Situationen bewiesen werden — sie werden geklammert und „`out`“ vorangestellt — und die *Trigger* — sie erhalten entsprechend das Schlüsselwort „`trig`“.

Denjenigen Relationen, die sich auf andere Situationen (und Situationstypen) beziehen, werden deren Bezeichner in der Beschreibung vorangestellt; es ergibt sich die Form

Situation : *Situationstyp Relation*

Dies ist bei Triggern der Nachbedingung und bei in anderen Situationen zu beweisenden Relationen der Fall.

Beispielhaft werden hier die verschiedenen Typen aufgeführt:

- Normale Relation: `is_owner (file1, stefan)`
- Negation: `not (is_owner (file2, stefan))`
- In anderer Situation zu beweisende Relation:
  `out (window2:fileview_window is_owner (fileFromOtherDir, X))`

```
facts

     type
     Globale Fakten des jeweiligen Situationstyps
     end

     Name der ersten Situation
     Fakten der ersten Situation
     endsit

     Name der zweiten Situation
     Fakten der zweiten Situation
     endsit

     ...
end
```

Abbildung 4.3: *Die Syntax zur Beschreibung der den einzelnen Situationen zugeordneten Fakten*

- Trigger: `trig (display (X))` (in der Vorbedingung)
  oder: `trig (window2:fileview_window display (X))` (in der Nachbedingung; mit Spezifikation der Zielsituation)

Abgeschlossen wird jede Aktionsregel durch ein Semikolon.

Nebenbedingungen (Constraints)

Nebenbedingungen sind durch die logischen Operatoren `and` (Konjunktion), `or` (Disjunktion) und `impl` (Implikation) verknüpfte Relationen, die negiert sein können. Auch sie werden jeweils durch Semikolon abgeschlossen. Nebenbedingungen werden normalerweise Beziehungen zwischen Fakten der aktuellen Situation beschreiben; sie können aber auch Relationen enthalten, die in anderen Situationen bewiesen werden.

Fakten

Fakten sind nichtnegierte Relationen. Es gibt keine negierten Fakten, da fehlende Fakten als falsch angesehen werden (Negation durch Nichtbeweisbarkeit; *negation–as–failure*). In diesem Abschnitt werden zusammen mit den Fakten die Situationen definiert, die zu Beginn vorhanden sind. Deren Anfangszustand wird durch die hier enthaltene Faktenmenge festgelegt; die Faktenmenge zerfällt innerhalb der Beschreibung in entsprechend viele Gruppen. Eine dieser Gruppen enthält die Fakten, die für alle Situationen des jeweiligen Typs gelten. Diese Fakten sind — wie die speziell für die Situationen definierten Fakten — den Situationen zugeordnet und werden jeweils unabhängig behandelt; beispielsweise kann ein solches Fakt in jeder Situation einzeln gelöscht werden. Die Möglichkeit, Fakten für den gesamten Situationstyp festzulegen, stellt damit eine notationelle Vereinfachung in der Spezifikationssprache dar.

Die syntaktische Struktur einer SUSI–Beschreibung ist in Abbildung 4.4 noch einmal graphisch dargestellt. Die vollständige Grammatik findet sich im Anhang A auf Seite 89. Beispiele für einfache SUSI–Spezifikationen stehen in den Anhängen B und C auf den Seiten 91 bzw. 95.

Schnittstellenbeschreibung

Zusätzlich zu der eigentlichen SUSI–Spezifikation ist noch eine Beschreibung der Schnittstellen erforderlich. Diese erfolgt in einer eigenen Datei, in der Relationen deklariert werden, die semantische Aktionen in der Anwendung auslösen oder Seiteneffekte auf die Präsentationsebene haben.

Im Fall der semantischen Aktionen der Anwendung werden einzelnen Relationen parametrisierbare Zeichenketten zugeordnet, die bei der Auswertung der Relation an die Anwendung geschickt werden bzw. (bei Implementierungen in interpretierten Sprachen) ein Stück Code darstellen, das ausgewertet werden kann (1). Relationen, die auf die Präsentationsebene wirken, erhalten zwei Zeichenketten; eine für den positiven und eine für den negativen Fall (2). Wird die Relation als Fakt etabliert, so wird die erste Zeichenkette an die Präsentationsschicht geschickt bzw. ausgewertet; wird sie gelöscht, die zweite Zeichenkette.[2] Zuvor werden noch Parameter in der Zeichenkette eingefügt; in der Definition können dafür Stellen markiert werden, an denen die Argumente der dazugehörigen Relation der Reihenfolge nach eingesetzt werden.

Die Definitionen werden in einer eigenen Datei abgelegt und von den Schlüsselwörtern `declarations` und `end` eingeschlossen. Sie haben folgende Form:

(1) *Relation* `:` `semantic` *ParametrisierteZeichenkette* `;`

(2) *Relation* `:` `presentation` *ParametrisierteZeichenkette* `:` *ParametrisierteZeichenkette* `;`

Genauer werden die Schnittstellen im Kapitel 5 beschrieben. Beispiele für vollständige Schnittstellenspezifikationen finden sich ebenfalls in den Anhängen B und C.

4.3 Die Architektur des Systems

In diesem Abschnitt soll ein Architekturansatz für SUSI dargestellt werden. Dabei werden die einzelnen Teile und ihr Zusammenwirken beschrieben. Der Schwerpunkt liegt dabei auf abstrakten Gesichtspunkten; eine Darstellung der konkreten Realisierung ist im Abschnitt 5.1 enthalten.

Der Programmablauf (d.h. die Operationen an der Benutzungsoberfläche) wird auf die im vorigen Abschnitt dargestellten Weise beschrieben. Die Spezifikation wird zunächst durch einen *Parser* eingelesen und in geeignete Datenobjekte transformiert. Damit wird die Benutzungsschnittstelle dann durch einen *Interpretierer* in der beschriebenen Weise gesteuert. Dieser führt die spezifizierten Aktionen aus und interagiert dabei mit der Anwendungsschnittstelle und der Präsentationsebene (Abbildung 4.5).

Der Parser wird dabei mit herkömmlichen Methoden des Übersetzerbaus realisiert (siehe z.B. [Waite, Goos 1984]). Er liest — gemäß der im Abschnitt 4.2 (und im Anhang A) beschriebenen

[2] Die Relationen der ersten Gruppe werden in den Vorbedingungen einer Regel wirksam; sie werden in der Sicht von SUSI *abgeleitet* und müssen daher einen entsprechenden Wert (z.B. *wahr* oder *falsch*) zurückgeben. Die Relationen der zweiten Gruppe werden in der Nachbedingung wirksam; da hier nur Fakten etabliert (bzw. gelöscht) werden und keine Ableitung stattfindet, ist ihr Rückgabewert belanglos. Treten diese Relationen in der Vorbedingung auf, so werden sie wie herkömmliche Relationen bewiesen.

Abbildung 4.4: *Graphische Darstellung der syntaktischen Struktur einer* SUSI*–Beschreibung. Rechts stehen Beispiele für die jeweiligen Beschreibungselemente.*

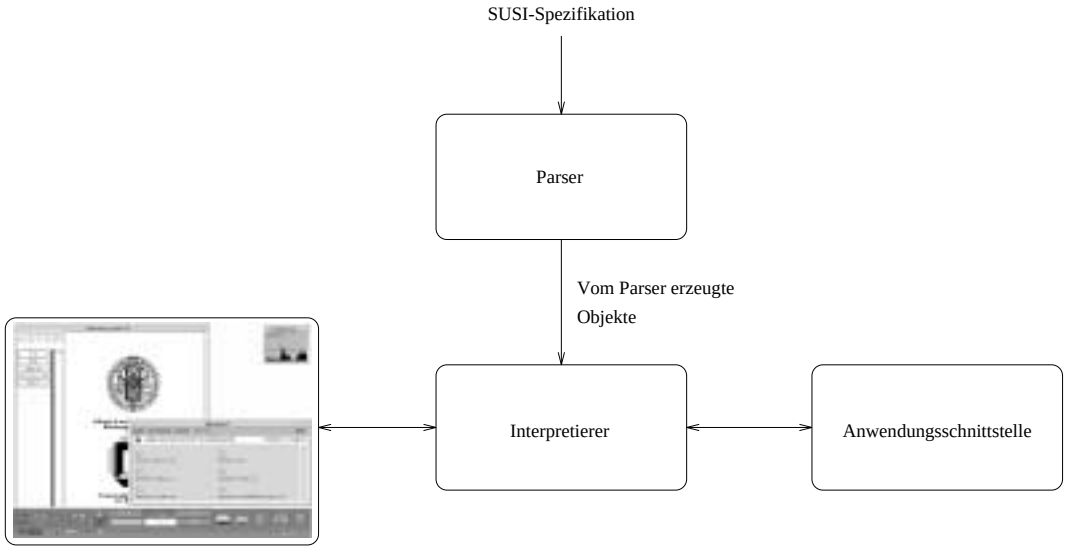

Abbildung 4.5: *Die Grobstruktur von* SUSI. *Hauptelemente sind der Parser, der Interpretierer für die Spezifikation und die Schnittstellen zu Darstellungsschicht und Anwendung.*

Syntax — die Definitionen für die Schnittstellen sowie die Aktionsregeln, Nebenbedingungen und Fakten ein.

Auf der Basis dieser Definitionen steuert der Interpretierer den weiteren Ablauf. Er interagiert dabei mit den Schnittstellen der Präsentationsschicht und der Anwendung, indem er Trigger und semantische Aktionen (*Callbacks*) absetzt und Funktionen zum Beweis von Relationen aufruft (die wiederum Seiteneffekte haben können). Die verschiedenen Ebenen bei der Steuerung sind aus Abbildung 4.6 ersichtlich.

Die Abbildung zeigt die Vorgänge beim Kopieren bzw. Verschieben von Dateien durch Ziehen des dazugehörigen Sinnbilds mit der Maus in ein anderes Fenster oder auf das Sinnbild eines Ordners. Der Vorgang des Ziehens löst das Absetzen eines Triggers an den Interpretierer von SUSI aus. Das Ursprungsfenster der Aktion ist auf dieser Ebene durch eine Situation repräsentiert, in der die notwendige Information über Sinnbild und dazugehörige Datei vorliegt.

Nach Maßgabe des Triggers wird nun die auszuführende Aktionsregel ausgewählt. Diese enthält die Bedingungen, die für die Operation erfüllt sein müssen; diese werden abgeleitet.

Ist die Ableitung erfolgreich, so wird eine semantische Aktion (in unserem Fall der UNIX–Befehl `mv` bzw. `cp`) an der Anwendungsschnittstelle abgesetzt. Diese ist mit einer Relation verbunden; kann sie fehlerfrei ausgeführt werden, wird als Ergebnis der Ableitung dieser Relation *wahr* zurückgegeben.

Nun werden in der mit dem Zielverzeichnis assoziierten Situation Fakten etabliert, die Information über die neu vorhandene Datei repräsentieren. Zuletzt wird — als Seiteneffekt dieser Etablierung — das dazugehörige Dateisymbol dargestellt.

Da in Fenstersystemen i.d.R. mehrere Fenster gleichzeitig geöffnet (d.h. mehrere Situationen gleichzeitig vorhanden) sind, bietet es sich an, für die verschiedenen Situationen jeweils eigene Prozesse zu aktivieren. Diese Prozesse kommunizieren, indem sie — beispielsweise über *Streams* — Trigger austauschen. Für jede Situation existiert dann eine eigene Instanz des Interpretierers. Das Generieren einer neuen Situation ist mit dem Starten eines neuen Prozesses verbunden.

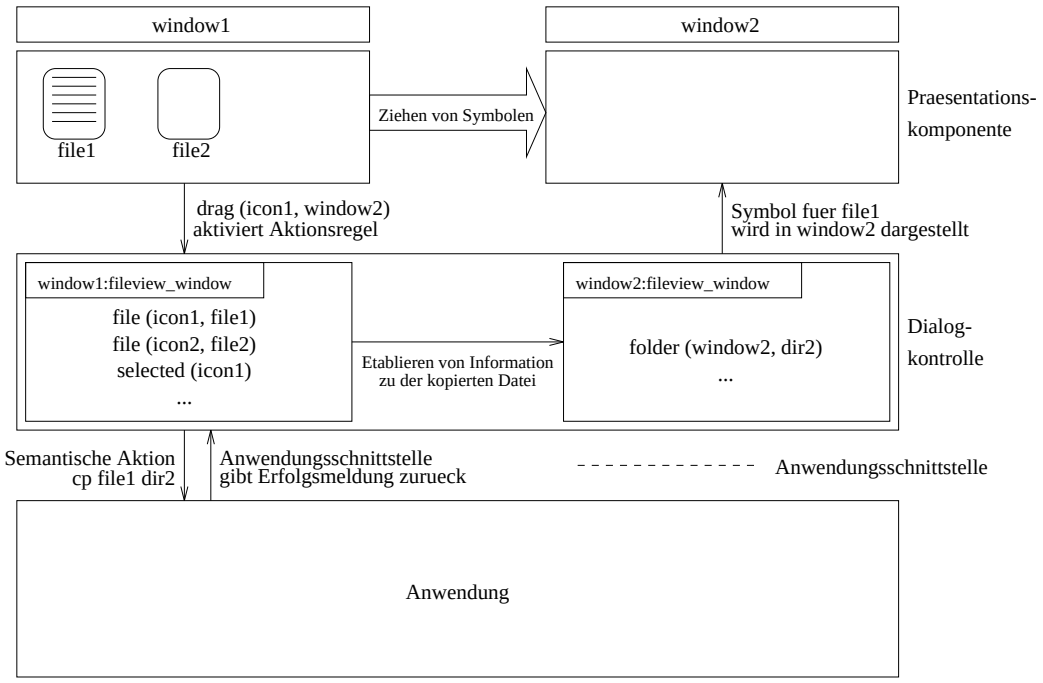

Abbildung 4.6: *Der Informationsfluß von Benutzereingabe bis zum Aufruf einer Anwendungsprozedur. Fakten, wie sie in den Situationen der mittleren Ebene dargestellt sind, können auch aus der Anwendung gelesen werden.*

Aus logischer Sicht stellen die Übergänge zu Oberfläche und zur Anwendung einen Bruch dar. Dieser Bruch wird beseitigt, indem die Effekte, die durch die SUSI–Steuerung auf diese beiden Systemteile bewirkt werden, so dargestellt werden, daß sie aus der Sicht von SUSI logischen Ableitungen entsprechen. Die Schnittstellen werden deswegen auf der Architekturebene als (virtuelle) Situationen modelliert, d.h. der Informationsaustausch an diesen Schnittstellen entspricht dem zwischen Situationen innerhalb der SUSI–Komponente. Die sich daraus ergebende Architektursicht zeigt Abbildung 4.7.

4.4 Vergleich von SUSI mit anderen Ansätzen

Der Ansatz von SUSI, wie er in diesem Kapitel beschrieben wurde, ist nach der Klassifizierung von Olsen ([Olsen 1992]) im Bereich der *produktionsbasierten* Ansätze zur Realisierung der Dialogkomponente anzusiedeln. Der Dialog wird durch Regeln (Produktionen) gesteuert, die nach dem Verfahren der Vorwärtsverkettung ausgeführt werden. Die Auswahl erfolgt dabei, indem eine ausgezeichnete Relation der Regel mit einem Trigger unifiziert wird; dabei ist eine einfache Form der Parameterübergabe durch die im Rahmen der Unifikation erfolgende Instantiierung möglich.

Die Ausführung von Regeln hängt dabei von der Herleitbarkeit einer Menge von Vorbedingungen aus einer Wissensbasis ab. Die Elemente der Wissensbasis sind so strukturiert, daß komplexe Objekte der Oberfläche (wie z.B. Fenster) jeweils eine eigene Faktenmenge besitzen.

Den Ansatz zeichnet besonders die Möglichkeit der Formulierung abstrakter Konzepte in Form von Nebenbedingungen aus: prädikatenlogischer Aussagen, die komplexe Zusammenhänge zwi-

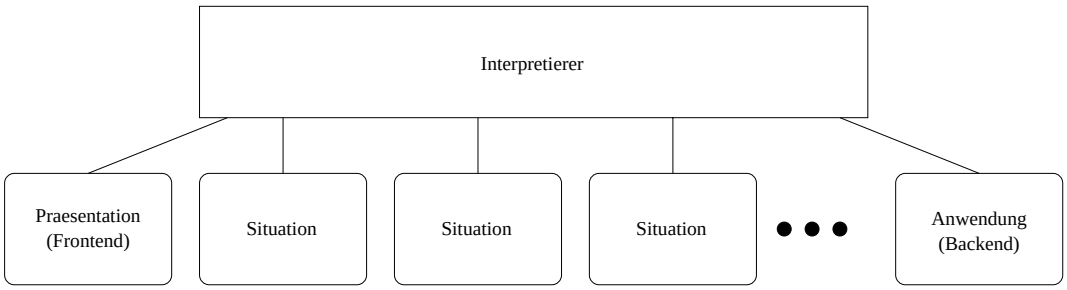

Abbildung 4.7: *Logischer Aufbau des* SUSI*–Systems. Präsentationsebene und Anwendung werden als spezielle Situationen modelliert.*

schen den Elementen der Wissensbasis ausdrücken. Außerdem kann das repräsentierte Wissen — durch Situationen — strukturiert werden; komplexen Objekten wie z.B. Fenstern ist jeweils eine eigene Situation zugeordnet; die Erzeugung kann dynamisch zur Laufzeit erfolgen.

Das Beispiel 5.2.2 (Seite 79) zeigt die Formalisierung eines komplexen Zusammenhangs: eine Datei kann dann in ein Verzeichnis verschoben werden, wenn die aktuelle Benutzerin bzw. der aktuelle Benutzer sowohl die Datei als auch das Verzeichnis besitzt.

Wie komplex die Objekte letztendlich sind, die in einer Situation zusammengefaßt werden, ist implementierungsabhängig; prinzipiell ist es beispielsweise auch denkbar, für jedes einzelne Sinnbild eine eigene Situation zu definieren. Ob dies sinnvoll ist, wird jedoch bezweifelt; der Kommunikationsaufwand mit anderen Situationen wäre bei einer so feinen Einteilung beträchtlich.

Durch die Formalisierung von Bedingungen zur Ausführung von Aktionsregeln und das in der Situation repräsentierte Wissen kann kontextabhängige Hilfe bei der Benutzung angeboten werden. Das Vorliegen der Voraussetzungen kann geprüft, im Fall des Scheiterns können Gründe — nicht ableitbare Bedingungen — angegeben werden.

In den folgenden Abschnitten soll nun ein Vergleich des Ansatzes von SUSI zu anderen produktionsbasierten Ansätzen, die im Kapitel 3 vorgestellt wurden, gezogen werden.

4.4.1 Sassafras

Die im SASSAFRAS–System vorgesehene Ableitungsmethode basiert auf *Ereignissen* und *Flags*. Diese Konzepte korrespondieren mit den in SUSI vorgesehenen *Triggern* und den einzelnen *Relationen der Vorbedingung*. Hier entfällt jedoch die Möglichkeit, entsprechend der ϵ–Regeln in SASSAFRAS Regeln ohne Trigger zu formulieren.

Geringere Ausdrucksmöglichkeiten ergeben sich bei SASSAFRAS durch den Verzicht auf ein– oder mehrstellige Relationen außerhalb von Ereignissen und das Fehlen der Möglichkeit, komplexe Sachverhalte in Form von Nebenbedingungen auszudrücken. Die Ableitung reduziert sich auf das reine Suchen von Flags in einer Datenbasis.

Weitgehende Ähnlichkeiten bestehen bezüglich der Architektur, was beim Vergleich der Abbildungen 3.3 (Seite 31) und 4.7 (Seite 61) deutlich wird. Einzelne ERL–Module sind den in SUSI vorgesehenen Situationen ähnlich, können aber im Gegensatz dazu nicht dynamisch zur Laufzeit erzeugt werden sondern werden bei der Implementierung der Benutzungsschnittstelle festgelegt.

4.4.2 Propositional Production System

Eine gegenüber SASSAFRAS erhöhte Ausdrucksmächtigkeit bietet das *Propositional Production System* (PPS). Als Bedingung für die Ausführung von Aktionsregeln sind — statt einzelner Flags — *Zustandsvektoren* vorgesehen, die nicht nur 2–wertig (gesetzt/nicht gesetzt) sind, sondern eine beliebig festzulegende Wertigkeit haben können (je nach der Anzahl der im *Zustandsraum* vorgesehenen Elemente für eine gegebene Vektorposition).

Aktionsregeln werden gemäß der Verträglichkeit der Vorbedingung mit dem momentanen Zustandsvektor aktiviert. Die Verbindung zur Präsentationsschicht stellt eine ausgezeichnete Klasse von Vektorelementen her, die aufgrund von Ereignissen an der Oberfläche gesetzt werden. Ähnliche Klassen von Vektorelementen gibt es für Abfragen, wie beispielsweise in Dialogboxen, und semantischen Aktionen. Hier besteht eine Ähnlichkeit zu den prozedurealen Fakten von SUSI.

Eine in der bestehenden Implementierung von SUSI noch nicht vorhandene (für einen späteren Zeitpunkt jedoch vorgesehene) Möglichkeit, die automatische Ermittlung aktiver bzw. inaktiver Aktionen wird in PPS durch den Vergleich der Vorbedingungen mit dem aktuellen Zustandsvektor realisiert.

Weitergehende Konzepte wie ein Mechanismus für die Übergabe von Argumenten an Regeln oder die Formulierung komplexer Konzepte als Nebenbedingung sind in PPS nicht vorgesehen. Über die Architektur eines so realisierten Systems geht aus der Literatur nichts hervor; es werden Überlegungen zur effizienten Implementierung durch Repräsentation der Bedingungen als Bitvektoren angestellt.

4.4.3 User Interface Design Environment

Ein weiter fortgeschrittenes Konzept, das auch Ansätze zur Erzeugung kontextorientierter Hilfe umfaßt, ist im *User Interface Design Environment* (UIDE) verwirklicht. Hier werden Bildschirmobjekte (wie z.B. Knöpfe (*buttons*)) an Vor– und Nachbedingungen gekoppelt, die beschreiben, ob sie aktivierbar sind und was sie bewirken.

Die Möglichkeit, Relationen mit Argumenten zu versehen, ist dabei gegeben. Komplexere Aussagen können aber auch in UIDE nicht gemacht werden. Die immer an einzelne Bildschirmprimitive gebundenen Ausführungsregeln bzw. Interaktionstechniken auf der Schnittstellenebene lassen auch keine Beschreibung komplexerer Objekte (wie in SUSI z.B. gesamte Fenster mit allen Unterobjekten) zu. Im Zuge der Weiterentwicklung wird jedoch derzeit eine Aufteilung in Anwendungs– und Schnittstellenebene vorgenommen. Eine Modularisierung der Faktenbasis ist noch nicht vorgesehen; in [Gieskens, Foley 1991] wird auf diesen Mangel hingewiesen.

Kontextabhängige Hilfe — bezogen auf die einzelnen Bildschirmprimitive — wird von dem System zur Verfügung gestellt. Die vorgesehenen Hilfstexte beschreiben zu den einzelnen Objekten, ob sie gerade aktivierbar sind und, wenn nicht, welche Aktionen zu ihrer Aktivierbarkeit führen.

4.4.4 Zusammenfassung

Der Vergleich mit den beschriebenen Systemen zeigt, daß SUSI in einigen Punkten eine konzeptuelle Erweiterung dieser Ansätze darstellt. Insbesondere die flexible Darstellung von Zusammenhängen durch Nebenbedingungen, der Ableitungsmechanismus mit Instantiierung von Variablen sowie die Modularisierung in dynamisch erzeugbare Situationen haben in den zum

Tabelle 4.1: *Gegenüberstellung einiger Eigenschaften von* SASSAFRAS, PPS, UIDE *und* SUSI

	SASSAFRAS	PPS	UIDE	SUSI
Ableiten von Vorbedingungen	Suche nach Flags	Vergleich mit Zustandsvektor	Fakten in CSB	Nebenbedingungen und Fakten
Formulierung komplexer Zusammenhänge				Nebenbedingungen
Wissensstrukturierung (Modularisierung)	statisch			Situationen (dynamisch)
Vorberechnung ausführbarer Regeln			ja	vorgesehen
Mechanismus zur Parameterübergabe	bei Ereignissen		ja	Unifikation mit Trigger
Kontextabhängige Hilfe			ja	ja

Vergleich herangezogenen Systemen in dieser Form keine Entsprechung. In Tabelle 4.1 sind einige Eigenschaften der einzelnen Ansätze noch einmal gegenübergestellt.

4.5 Zwei Beispiele

Die in diesem Kapitel vorgestellten Konzepte sollen nun anhand von zwei ausführlichen Beispielen verdeutlicht werden.

In den vorigen Abschnitten wurde bereits mehrfach das Beispiel des Dateimanagers erwähnt, wie er in den meisten heutigen Systemen üblich ist (Beispiele 4.1.1–4.1.4). In diesen Dateimanagern sind Dateien und Verzeichnisse als Sinnbilder repräsentiert, die in Fenstern dargestellt werden. Die Interaktion mit dem System erfolgt i.d.R. durch Verschieben von solchen Sinnbildern (direkte Manipulation). Außerdem gibt es Pull–Down–Menüs, in denen Befehle ausgewählt werden können, die teilweise auf das ganze Fenster wirken (z.B. ein Verzeichniswechsel) und teilweise bestimmte Dateiobjekte ansprechen, deren Sinnbild zuvor selektiert wurde.

Das von SUSI gesteuerte Selektieren einer Datei wird durch einen Trigger (z.B. `click ('file1')`) ausgelöst, der beispielsweise durch das Anklicken des Sinnbilds einer Datei erzeugt wird. Nun wird — durch Unifizieren mit den Triggerrelationen der vorhandenen Regeln und unter Berücksichtigung der Auswahlfunktion — eine anwendbare Regel ausgewählt, hier etwa

```
fileWaehlen
  trig (click (X)), not (selected (X)) --> selected (X);
```

Durch die Unifikation wird dabei `X` durch `'file1'` substituiert. Anschließend werden die zweite Vorbedingung abgeleitet. Die Relation `not (selected (X))` (hier also `not (selected ('file1'))`) prüft, ob die Datei bereits selektiert ist; ist sie es, so wird die Aktion nicht ausgeführt und nach einer weiteren Regel gesucht. Ansonsten ist die Vorbedingung damit abgeleitet. Nun werden die Nachbedingungen der — ausführbaren — Aktionsregel behandelt. Da

der Zustand `selected` an der Oberfläche sichtbar sein soll, wird ein Seiteneffekt an der Präsentationsschicht ausgelöst. Die Relation `selected` wird man deswegen als prozedurale Relation definieren; der `declarations`–Teil enthält etwa folgende Zeile:

```
selected () : presentation
   "(select-icon \"#1\")" : "(unselect-icon \"#1\")";
```

Dabei sind `select-icon` und `unselect-icon` LISP–Funktionen, die bei Etablierung bzw. Entfernung des Fakts zu einer entsprechenden Darstellung des Sinnbilds (z.B. invertiert/nicht invertiert) führen. Der Name des Sinnbilds (also `'file1'`) wird vor dem Aufruf an der Stelle `#1` eingesetzt. In diesem Fall führt dann der Aufruf der Funktion `select-icon` dazu, daß das Sinnbild beispielsweise invertiert dargestellt wird. Zusätzlich wird SUSI die Datei als selektiert bekannt gemacht, indem das Fakt `selected ('file1')` etabliert, d.h. der Faktenbasis zugefügt wird.

Etwas komplexer wird es, wenn ein Sinnbild mit der Maus auf einen Ordner gezogen wird. Hier sind prinzipiell zwei Aktionen denkbar:

- Die Datei soll kopiert (UNIX–Befehl `cp`) werden.
- Die Datei soll verschoben (UNIX–Befehl `mv`) werden.

Dafür, daß eine solche Aktion durchgeführt werden kann, gibt es einige Voraussetzungen. Beispielsweise muß es sich um eine Datei handeln; das Ziel muß ein Verzeichnis sein. Ob eine Datei kopiert oder verschoben wird, kann beispielsweise davon abhängen, ob die betreffende Datei der aktuellen Benutzerin bzw. dem aktuellen Benutzer gehört. Diese Abhängigkeiten können durch Nebenbedingungen formalisiert werden. Die Bedingung, daß eine Datei verschoben wird, kann z.B. formalisiert werden als

```
file (A) and folder (B) and currentUser (C) and
owns (A, C) and owns (B, C)
impl
movable (A, B);
```

Eine Aktionsregel, die das Verschieben einer Datei bewirkt, ist dann

```
dragInVerzeichnisAndMove
  trig (drag (Y, X)), movable (Y, X), move (Y, X) --> ;
```

Zunächst wird — durch das Verschieben an der Oberfläche — ein Trigger, erzeugt, z.B. `drag ('file1', 'folder2')`. Anschließend wird `movable` durch die oben gezeigte Nebenbedingung abgeleitet.

Der Aufruf der semantischen Aktion, die das Verschieben an der Anwendungsschnittstelle bewirkt, ist ein Seiteneffekt der Relation `move` dieser ist — wie oben `selected` — im Deklarationsteil festgelegt; die Definition lautet

```
move () : semantic "(rename-files \"#1\" \"#2\")";
```

Hier wird ein Wert zurückgegeben: *wahr*, wenn die Operation erfolgreich ausgeführt wurde, ansonsten *falsch*. Die Nachbedingung ist leer; die Aktion ist damit nach der semantischen Aktion beendet.

Die vollständige Spezifikation, aus der die Beispiele entnommen sind, steht im Anhang B.

5

Das System

In diesem Kapitel wird das im Rahmen der Arbeit realisierte System beschrieben. Dabei wird zunächst die vorliegende Implementierung vorgestellt. Anschließend wird die Arbeitsweise anhand des in SUSI spezifizierten Dateimanagers dargestellt und eine kurze Anleitung zur Beschreibung von Benutzungsoberflächen gegeben. Zuletzt wird eine erste Bewertung der Implementierung im Hinblick auf die Performanz vorgenommen.

5.1 Die Implementierung

Eine Implementierung des beschriebenen Systems zur Dialogkontrolle wurde auf einem Arbeitsplatzrechner von Hewlett–Packard unter Verwendung von Common–Lisp und CLOS (Common–Lisp Object System) vorgenommen ([Steele 1990], [Franz 1992], [Keene 1989]). Bei einigen wenigen Funktionen für die vorliegende Anwendung wurde auf C ([Hewlett–Packard 1991]) zurückgegriffen. Die Elemente der Oberfläche wurden mit Hilfe von CLIM (Common–Lisp Interface Manager) im Rahmen der Arbeit von Winter ([Winter 1994]) implementiert.

5.1.1 Der Eingabeparser

Die syntaktische Struktur einer SUSI–Spezifikation wurde bereits im Abschnitt 4.2 beschrieben; die Grammatik findet sich im Anhang A.

Zum Einlesen der Spezifikation wurde ein LL–Parser implementiert, der mit rekursivem Abstieg arbeitet. Er liest zwei Dateien ein:

- Die erste Datei — mit der Endung „`.decl`“ — enthält die Schnittstellenspezifikation, d.h. die Festlegung der prozeduralen Fakten und die an die Schnittstellen zu Präsentationsebene bzw. Anwendung zu übermittelnden Zeichenketten.
- Die zweite Datei — mit der Endung „`.susi`“ — enthält die eigentliche Dialogbeschreibung: Aktionsregeln, Nebenbedingungen und ggf. Fakten.

Beim Einlesevorgang werden unterschiedliche CLOS–Objekte erzeugt. Auf der höchsten Ebene gibt es Objekte für Situationen; sie haben als Typ einen Situationstyp (wofür zur Laufzeit jeweils eine Klasse erzeugt wird). Ihre Slots enthalten dann Aktionsregeln, Nebenbedingungen und Fakten. Diese wiederum enthalten in ihren Slots — entsprechend ihrem Aufbau — Listen von Relationsobjekten: Aktionsregeln enthalten beispielsweise je eine Liste von Objekten für die Vor– und für die Nachbedingung.

Zur Erzeugung von Situationen zur Laufzeit wird eine Instanz der Objektklasse des jeweiligen Situationstyps generiert. Diese Instanz erbt von der Klasse die dort festgelegten Nebenbedingungen und Aktionsregeln. Der Zusammenhang zwischen den einzelnen Klassen ist in Abbildung 5.1 dargestellt.

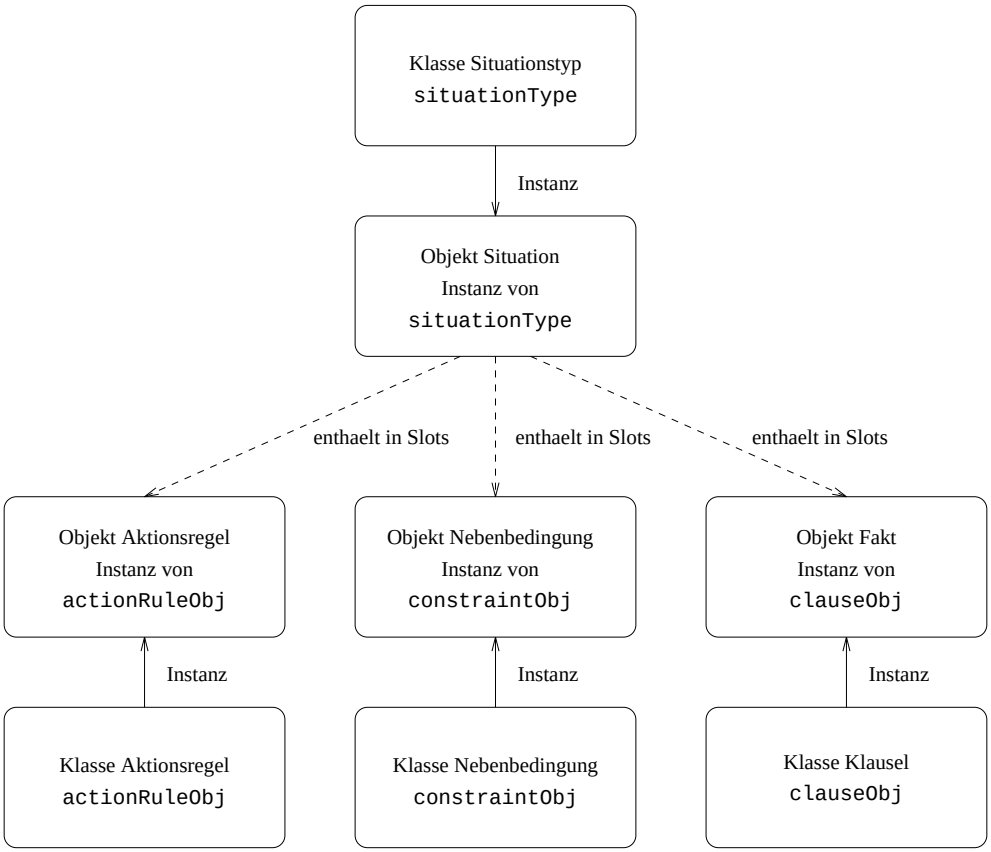

Abbildung 5.1: *Objektklassen als Teile einer Situationsdefinition. Die Objekte für die Beschreibungselemente sind in den Slots der Situationsobjekte enthalten.*

Für Relationen (Prädikate) sind zunächst zwei Oberklassen definiert, die allein zur Vererbung ihrer Slotdefinitionen bestehen, also Schablonen für die verschiedenen konkreten Relationsobjekte bilden:

`praedikat` enthält die Slots, um eine gewöhnliche Relation zu spezifizieren: *Namen*, *Vorzeichen (Polarität)*, eine Liste der *Argumente* und die *Stelligkeit*.

`sit-mixin` fügt — bei Bedarf — Information über die zugeordnete *Situation* hinzu: neben dieser selbst ist in einem weiteren Slot der *Typ* spezifiziert. Diese Angaben werden von Relationen benötigt, die auf andere Situationen zugreifen.

Für die verschiedenen Typen von Relationen sind fünf unterschiedliche Objektklassen vorgesehen:

`internalPraed` besitzt die in `praedikat` definierten Slots. Das Objekt repräsentiert normale, innerhalb von SUSI zu beweisende Relationen.

`externalPraedSemantic` enthält zusätzlich den Anwendungsbefehl als Zeichenkette. Das Objekt repräsentiert semantische Aktionen (Anwendungsbefehle): Relationen, die zumeist in der Vorbedingung vorliegen und als Seiteneffekt eine Anwendungsoperation auslösen.

`externalPraedPresentation` entspricht der Klasse `externalPraedSemantic`, enthält aber zwei Befehle: für den Fall der Etablierung eines Fakts und für den Fall seiner Entfernung. Das Objekt repräsentiert prozedurale Relationen in der Nachbedingung (i.d.R. Operationen in der Präsentationsschicht); es erzeugt zu diesem Zweck ebenfalls Seiteneffekte. Erscheint eine solche Relation in der Vorbedingung, so wird sie wie die Relationen der Klasse `internalPraed` intern abgeleitet.

`outTriggerPraed` hat die Slots aus `praedikat` und aus `sit-mixin`. Zusätzlich müssen hier noch die Substitutionen für die Relation explizit vorliegen. Das Objekt repräsentiert Trigger.

`otherSitPraed` enthält ebenfalls die Slots aus `praedikat` und `sit-mixin`. Für Abfragen, z.B. aus der Anwendung, ist noch ein externer Befehl vorgesehen. Das Objekt repräsentiert Relationen, die in anderen Situationen abgeleitet werden.

Die Objektklassen werden bei der Initialisierung erzeugt; die Objekte entstehen während des Einlesens zu Beginn gemäß der SUSI–Dialogbeschreibung oder können dynamisch zur Laufzeit vom Interpretierer generiert werden, beispielsweise bei der Erzeugung einer neuen Situation oder eines neuen Fakts.

Schnittstellenspezifikation

In der Schnittstellenspezifikation werden einzelnen Relationen semantische Aktionen als Zeichenketten — in unserer Implementierung LISP–Funktionen — zugeordnet. Diesen können Parameter übergeben werden. Dazu werden Stellen markiert (`#1, #2, ...`), an denen die Parameter eingefügt werden sollen. Dies erfolgt dann gemäß der Reihenfolge der Argumente der Relation.

> Beispiel 5.1.1 *Ist einer Relation* `move/2` *die Zeichenkette* `(rename-file #1 #2)` *zugeordnet, so wird beim Auftreten eines Literals* `move (file, folder)` *der Funktionsaufruf* `(rename-file file folder)` *erzeugt.*

Dies geschieht zur Laufzeit gemäß der aktuellen Substitution; d.h. die Relationen können Variablen als Argumente besitzen. Diese müssen jedoch zum Zeitpunkt des Aufrufs instantiiert sein.

Die Parameter für die Aktionen werden im Zuge der Ableitung von Relationen der Vorbedingung ermittelt. Die endgültigen Substitutionen liegen dabei erst nach Ableitung der letzten Relation vor. Um zu vermeiden, daß Aufrufe erzeugt werden, die durch später zu beweisende Elemente der Vorbedingungen, die die Substitutionsmenge einschränken, noch verschwinden würden, sollten die semantischen Aktionen immer als letztes spezifiziert werden. Ansonsten besteht die Möglichkeit, daß falsche Befehle an die Anwendung abgesetzt werden.

SUSI–Dialogspezifikation

Beim Einlesen der Dialogspezifikation werden der Reihe nach Objekte für Aktionsregeln, Nebenbedingungen und Fakten erzeugt und in die entsprechenden Slots der Situationsobjekte eingefügt. Vorher wird eine Klassendefinition für den gerade eingelesenen Situationstyp und die Situationen als Objekte dieser Klasse generiert.

- Für die Aktionsregeln werden die Relationen in Vor– und Nachbedingung nacheinander gelesen und entsprechend in zwei Listen eingegliedert.
- Die Nebenbedingungen sind — in der derzeitigen Implementierung — Horn–Klauseln. Sie können aber als beliebige Formeln angegeben werden und durchlaufen nach dem eigentlichen Einlesen noch einen Transformationsprozeß, in dem sie auf Klauselform gebracht werden (zur Behandlung dabei entstehender Klauseln, die nicht die Horn–Eigenschaft besitzen siehe Abschnitt 4.1.3).
- Für die Fakten werden (aus Gründen der einheitlichen Darstellung) Klauselobjekte — mit leerem Rumpf — erzeugt.

5.1.2 Der Interpretierer

Der Interpretierer steuert das System, indem er entsprechend dem jeweiligen Trigger Aktionsregeln auf Ausführbarkeit testet und ggf. ausführt. Er arbeitet in drei Stufen:

1. Zunächst wählt er gemäß des — global definierten — Triggers eine anwendbare Aktionsregel aus. Das Auswahlkriterium ist dabei die Unifizierbarkeit des Triggers mit der entsprechenden Relation.
2. Nun versucht er, durch Resolution die einzelnen Relationen der Vorbedingung der ersten ausgewählten Regel zu beweisen. Für jede Relation wird dabei zunächst eine Liste durchsucht, die bereits abgeleitete Relationen enthält. Ist diese leer, so wird eine Liste der anwendbaren Nebenbedingungen (Horn–Klauseln) und Fakten (die als Horn–Klauseln mit leerem Rumpf repräsentiert sind) erstellt und abgearbeitet. Dabei werden alle möglichen Substitutionen in einem Durchlauf erzeugt; die Notwendigkeit eines Zurücksetzens (*backtracking*) entfällt dadurch. Die abgeleiteten Relationen werden dann wieder zwischengespeichert.
3. Konnte bei einer Regel die Vorbedingung vollständig abgeleitet werden, wird zur Behandlung der Nachbedingung übergegangen. Für die positiven Literale werden Fakten erzeugt und in die Faktenliste der Situation eingegliedert; mit den negativen Literalen unifizierbare Fakten werden aus der Liste entfernt. Zusätzlich wird die Liste der bereits abgeleiteten Relationen gelöscht, da möglicherweise ihr Inhalt durch die Änderung der Faktenbasis nicht mehr gültig ist.

Schlägt bei einer anwendbaren Aktionsregel die Ableitung der Vorbedingung fehl (d.h. ist die Regel nicht ausführbar), so wird gemäß der Auswahlfunktion die nächste anwendbare Regel — in unserem Fall die im Text folgende — selektiert.

Auswahl der Aktionsregel

Der Mechanismus für die Auswahl der anzuwendenden Regel ist naheliegend: Die Regeln werden von oben nach unten daraufhin untersucht, ob ihre Triggerrelation in der Vorbedingung mit dem aktuellen Trigger unifizierbar ist. Ist dies bei einer Regel der Fall, so ist sie anwendbar und es wird zum Beweis der Vorbedingung übergegangen. Ist diese (als Ganzes) nicht beweisbar, d.h. die Aktionsregel nicht ausführbar, so wird die nächste anwendbare Regel ausgewählt. Existiert keine weitere solche Regel, so wird der nächste Trigger bearbeitet.

Die triviale Auswahlstrategie zeichnet sich durch zwei Vorzüge aus: da keine komplexe Auswahlfunktion berechnet werden muß, ist das Verfahren effizienter als solche, die weitergehende Kriterien für die Auswahl heranziehen. Außerdem ist es flexibel in dem Sinn, daß bei der Programmierung eine implizite Prioritätenvergabe (durch entsprechende Anordnung der Regeln) möglich ist.

Beweis der Vorbedingungen

Wie bereits erwähnt, gibt es drei Kategorien von Relationen in der Vorbedingung, die auf unterschiedliche Weise bewiesen werden:

- Normalerweise wird eine Relation innerhalb der aktuellen Situation (d.h. mit deren Nebenbedingungen und Fakten) bewiesen. Sie wird zu diesem Zweck an eine Auswertefunktion übergeben.
- In manchen Fällen ist Information aus anderen Situationen notwendig. Dann wird die Relation in dieser Situation (also mit deren Nebenbedingungen und Fakten) bewiesen; der Ablauf ist aber der gleiche wie im ersten Fall. Das Relationsobjekt muß dabei die Namen von Situation und Situationstyp enthalten.
- Teilweise ist der Zugriff auf Information hauptsächlich der Anwendung (und manchmal auch der Darstellungsebene) notwendig. Die wird durch Schnittstellenfunktionen realisiert, die die gleichen Datenstrukturen zurückliefern wie die Auswertefunktion des Interpretierers. Auch Kommandos an die Anwendung (semantische Aktionen) werden durch einen entsprechenden Funktionsaufruf realisiert. Hierauf wird im Abschnitt 5.1.3 näher eingegangen.

Die Auswertefunktion des Interpretierers wird mit der zu beweisenden Relation aufgerufen und arbeitet rekursiv gemäß Algorithmus 4.1.1. Zunächst werden die Fakten der Situation untersucht. Bei Unifizierbarkeit werden die entsprechenden Substitutionen erzeugt.[1] Wird die Relation mit Hilfe von Nebenbedingungen bewiesen, so wird — mit den Elementen des Rumpfs als neuen zu beweisenden Relationen — rekursiv fortgefahren, bis alle Bedingungen durch Fakten ableitbar sind (bzw. nicht abgeleitet werden können). Bei der Ableitung werden in einem Durchlauf alle möglichen Belegungen berechnet.

Während der Ableitungen werden keine Konstanten propagiert. Alle Anfragen erfolgen mit Variablen (es sei denn, es sind bereits in der Spezifikation Konstanten in den Ausdrücken enthalten). Deswegen hat die Datenstruktur `subst`, in der die jeweils gültigen Substitutionen gehalten werden, besondere Bedeutung. Sie ist als Keller organisiert, auf den auf jeder Rekursionsebene die gerade aktuellen Substitutionen geschoben werden.

[1] Es ist hier zu beachten, daß Fakten wegen der geforderten Bereichsbeschränktheit immer Grundterme sind, d.h. keine Variablen enthalten.

Die Substitutionen auf jeder Ebene sind Listen von Listen der möglichen Belegungen. Lautet beispielsweise eine Anfrage

```
a (X, Y)
```

und ist aus Fakten und Nebenbedingungen

```
a (m, t)
a (n, t)
a (m, u)
```

ableitbar, so entsteht die Substitutionsliste

```
(((X m) (Y t)) ((X n) (Y t)) ((X m) (Y u)))
```

also einer Liste der Tupel, die eine Lösung der Anfrage bilden, jeweils indiziert durch die dazugehörigen Variablen.

Auf diesen Listen werden — zusammen mit der jeweils neu hinzukommenden Substitution — die Operationen *Konjunktion*, *Disjunktion* und *Subtraktion* ausgeführt. Dazu gibt es die Funktionen `setNewSubstAnd`, `setNewSubstOr` und `setNewSubstMinus`.

`setNewSubstAnd` wird bei konjunktiv verknüpften Relationen verwendet, also z.B. für die Ergebnisse innerhalb der Vorbedingung einer Regel oder im Rumpf einer Nebenbedingung. Ergebnis ist eine Liste der Substitutionen, die sowohl in der neu abgeleiteten als auch in der vorigen Substitutionsliste enthalten sind.

`setNewSubstOr` wird verwendet, um die verschiedenen Instantiierungen bei mehreren möglichen Lösungen zusammenzufassen. Als Ergebnis liefert sie eine Liste der Substitutionen, die entweder neu abgeleitet wurde oder vorher schon enthalten war.

`setNewSubstMinus` entfernt abgeleitete Substitutionen wenn die Anfrage negiert ist. Ergebnis sind die Substitutionen, die vorher bereits vorhanden waren und *nicht* neu abgeleitet wurden.

Das Ergebnis einer Anfrage ist dann eine Liste mit dieser Anfrage und den abgeleiteten Substitutionen. Das Ergebnis der Anfrage des obigen Beispiels wäre demnach (die Relation ist nach Lisp–Konvention dargestellt):

```
((a X Y) (((X m) (Y t)) ((X n) (Y t)) ((X m) (Y u))))
```

Notwendig wird dieses Vorgehen dadurch, daß bei der Ableitung *alle* Instantiierungen berechnet werden. Gibt es für eine Anfrage mehrere mögliche Substitutionen, so wäre sonst jede mögliche Instantiierung der Variablen der Relation abzuleiten — jeweils mit einer neuen Anfrage. Durch die verwendeten Listen kann die Anfrage dagegen für alle Substitutionen gleichzeitig erfolgen.

Sollen Kommandos an die Anwendung übergeben werden, so müssen die Relationen entfaltet werden. Soll etwa der Befehl

```
mv X Y
```

abgesetzt werden und besteht eine Substitution

```
(((X file1) (Y folder1)) ((X file2) (Y folder1)) ((X file3) (Y folder2)))
```

so werden durch eine Funktion `unfold` die Aufrufe

```
mv file1 folder1
mv file2 folder1
mv file3 folder2
```

erzeugt.

Werden derartige Aufrufe an die Anwendungsschnittstelle abgesetzt, so müssen die aufgerufenen Funktionen — um intern Einheitlichkeit zu gewährleisten — die gleichen Datenstrukturen zurückgeben. Die von der Anwendung gelieferten Daten müssen ggf. in der Schnittstellenfunktion angemessen aufbereitet werden. Schlägt ein Aufruf fehl (kann er z.B. nicht erfolgreich ausgeführt werden), so muß die Funktion als Ergebnis `nil` zurückgeben.

Verarbeitung der Nachbedingung

Ist die Vorbedingung vollständig (erfolgreich) abgearbeitet, wird zur Behandlung der Nachbedingung übergegangen. Diese besteht intern zunächst aus zwei Operationen:

- Für neu zu etablierende Fakten — repräsentiert als *positive* Literale — werden entsprechende Objekte erzeugt und in die Faktenbasis eingegliedert, d.h. in die entsprechende Liste eingefügt. Die Argumente der positiven Literale müssen entweder Konstanten sein, oder Variablen, die während der Ableitung der Vorbedingung instantiiert wurden. (Bei nicht instantiierten Variablen müßten sonst alle möglichen Substitutionen bei der Etablierung der Fakten berücksichtigt werden.)
- Ungültig werdende Fakten werden aus der Faktenbasis entfernt. Ungültig sind alle Fakten, die mit in der Nachbedingung auftretenden *negativen* Literalen unifizierbar sind. Im Gegensatz zu positiven Literalen können Variablen enthalten sein; dadurch können mehrere Fakten gleichzeitig entfernt werden.

Zusätzlich werden evtl. noch Trigger erzeugt, um in der Ausführung fortzufahren. Diese werden in die entsprechende Warteschlange eingefügt.

Einige ausgezeichnete Literale haben eine direkte Auswirkung auf die Darstellung. Mit ihnen sind deswegen zwei Zeichenketten (z.B. Funktionsaufrufe) assoziiert, die in der Deklarationsdatei (`*.decl`) definiert werden. Davon wird der eine bei der Etablierung des Fakts an die Darstellungsschicht abgesetzt, der andere beim Löschen.

> Beispiel 5.1.2 *Die an der Oberfläche darzustellenden Dateisymbole können beispielsweise durch ein Fakt* `fileIcon ()` *repräsentiert sein. Werden diese als Fakten etabliert, so ist der Präsentationsschicht mitzuteilen, daß ein neues Symbol dargestellt werden soll. Wird ein solches Fakt gelöscht, so ist entsprechend das Symbol zu entfernen.*

An der Schnittstelle zur Präsentationsschicht müssen dann, wie an der Schnittstelle zur Anwendung, Funktionen bereitgestellt werden, die die entsprechenden Operationen realisieren.

Zuletzt werden noch die zwischengespeicherten Relationen, die beim Beweis der Vorbedingung abgeleitet wurden, gelöscht, da sich durch die Änderungen in der Faktenbasis ihre Gültigkeit geändert haben kann.

5.1.3 Schnittstellen

Während das System zur Laufzeit durch den Interpretierer gesteuert wird, der innerhalb der gerade aktiven Situation Aktionsregeln ausführt, findet eine Kommunikation mit verschiedenen anderen Instanzen statt:

- Die verschiedenen Situationen kommunizieren *untereinander*. Dabei werden einerseits durch die aktuelle Situation Informationen aus anderen Situationen angefordert; andererseits werden Trigger abgesetzt, um Aktionen in anderen Situationen anzustoßen.
- Die Situationen kommunizieren mit der Anwendung (*Backend*) und der Präsentationsebene (*Frontend*). Benutzeraktionen an der Oberfläche werden — in interpretierter Form — an die Anwendung weitergeleitet; Änderungen in der Anwendung müssen an der Oberfläche dargestellt werden. Darüberhinaus müssen auch hier Informationen durch die aktuelle Situation abgefragt werden können.
- Außerdem muß eine Schnittstelle zum Hilfssystem bestehen; insbesondere muß es diesem ermöglicht werden, auf die innerhalb der Situationen repräsentierte Information zuzugreifen.

Für die Situationen innerhalb der logischen Beschreibung existieren die folgenden Möglichkeiten, Information auszutauschen:

- Aktionen in anderen Situationen werden durch Trigger aktiviert. In der Beschreibung wird jeder Trigger mit der Situation, für die er bestimmt ist, parametrisiert. Meistens wird dies die aktuelle Situation selbst sein; es kann auf diese Weise aber auch eine andere Situation angesprochen werden.
- Genauso, wie innerhalb einer Situation Relationen mit der eigenen Faktenbasis bewiesen werden können, kann auch auf die Information anderer Situationen zugegriffen werden. Die Relation wird in der angegebenen Situation bewiesen; anschließend wird (mit den dadurch errechneten Substitutionen) in der aktuellen Situation fortgefahren.

Realisiert wird die Kommunikation zwischen Situationen, für die jeweils ein eigener Prozeß generiert werden soll, durch Streams, über die geeignete Daten (z.B Trigger, Anfragen) übermittelt werden.

Um die Verbindungen zwischen interpretierter logischer Beschreibung und Front– bzw. Backend zu gewährleisten, müssen nun auch diese Schnittstellen in geeigneter Weise modelliert werden.

Der in dieser Arbeit eingeschlagene Weg ist folgender: Front– und Backend werden als zusätzliche Situationen (eines Situationstyps `extern`) aufgefaßt. Es werden Schnittstellenfunktionen bereitgestellt, die bei der Ableitung entsprechender Relationen aufgerufen werden und die die gleiche Datenstruktur zum Ergebnis haben wie die Auswertefunktion des Interpretierers. Der sich damit ergebende logische Aufbau wurde bereits in der Abbildung 4.7 (Seite 61) gezeigt.

> Beispiel 5.1.3 *Sollen aus einem Verzeichnis die darin enthaltenen Dateien gelesen werden, so wird eine Relation* `file (X)` *bewiesen, indem eine assoziierte Funktion an der Anwendungsschnittstelle aufgerufen wird. Enthält das aktuelle Verzeichnis die Dateien* `file1` *und* `file2`*, so wird diese Funktion als Ergebnis die möglichen Substitutionen zurückgeben, also die Liste* `((file X) (((X file1)) ((X file2))))`.

Ähnlich wird verfahren, wenn es darum geht, *semantische Aktionen* (entsprechend den *Callbacks* in anderen ereignisbasierten Systemen) in der Anwendung zu aktivieren. Es werden mit Relationen Funktionen assoziiert, denen die Argumente der Relation (in geeigneter Weise) mitgegeben werden. Ist die Ausführung der Funktion erfolgreich, so wird die Relation zu *wahr* ausgewertet; sonst wird sie zu *falsch* ausgewertet.

Beispiel 5.1.4 *Soll — z.B. bei einem Dateimanager — eine Datei kopiert werden, so wird eine Relation, etwa* `copy (file, folder)` *ausgewertet, indem eine assoziierte* LISP*–Funktion, etwa* `(run-shell-command ``cp file folder'')`*, ausgeführt wird.*

Ein andere Variante solcher *prozeduraler Fakten* ist im Bereich der Kopplung mit der Präsentationsebene vorgesehen. Darstellungen in der Präsentationsschicht sind abhängig vom Zustand in der Dialogsteuerung, d.h. von der momentanen Faktenmenge, bzw. von einem Teil der in ihr enthaltenen Fakten.

Dies bedeutet, daß zur Laufzeit eine Änderung der Menge bestimmter Fakten eine Änderung der Darstellung an der Oberfläche nach sich zieht. Werden z.B. im Verzeichnisfenster eines Dateimanagers Symbole für Dateien dargestellt, so existieren in der korrespondierenden Situation Fakten, etwa `fileIcon (...)`. Erscheint im Verzeichnis eine neue darzustellende Datei, so wird in der Faktenmenge der Situation ein neues Fakt etabliert. Diese Etablierung muß der Oberfläche bekannt gemacht werden; diese sorgt dann für die Darstellung.[2] Entsprechend verhält es sich, wenn eine Datei aus dem Verzeichnis verschwindet.

Aus diesem Grund werden solchen Fakten *zwei* Funktionen zugeordnet: eine, die bei der Etablierung, und eine, die bei der Eliminierung ausgeführt wird.

Ein weiteres Problem, das sich an der Schnittstelle zwischen Präsentationsebene und Dialogsteuerung stellt, ist die Realisierung semantischer Rückmeldungen, d.h. die Darstellung von Eigenschaften einzelner Objekte, die vom Zustand der Anwendung bzw. des Dialogs abhängen. Einzelne Menüpunkte können etwa in einem bestimmten Systemzustand inaktiv, d.h. nicht auswählbar sein. (Beispielsweise kann ein `open` zum Öffnen einer Datei nicht ausgeführt werden, wenn keine Datei ausgewählt ist.) Teilweise wurde dies direkt in der Darstellungsebene realisiert (beispielsweise durch verschiedene Datenstrukturen, die Objekte mit unterschiedlichen Eigenschaften aufnehmen). Eine Erweiterung durch Vorberechnung von Aktionsregeln und daraus resultierender Darstellung aktiver bzw. nicht aktiver Objekte (ähnlich wie in PPS) ist denkbar; für unmittelbare Rückmeldungen erscheint dies jedoch als zu zeitaufwendig.

Realisierung der Schnittstelle zur Anwendung

An der Schnittstelle zur Anwendung finden nach dem eben gesagten drei Arten der Kommunikation statt:

1. Es werden Semantische Aktionen in der Anwendung angestoßen, d.h. aufgrund von Ereignissen an der Oberfläche Anwendungsfunktionen ausgeführt. Dies wird realisiert durch die die Übergabe von LISP–Funktionen als Zeichenkette.

[2] Anders bei unsichtbaren Dateien, z.B. bei UNIX bei denjenigen, deren Dateiname mit einem Punkt beginnt.

2. SUSI muß auf Daten der Anwendung zugreifen können. Dies geschieht durch die Ableitung von Relationen mit Bereitstellung der Daten durch Instantiierung der Argumente. Das bedeutet, daß mit den Relationen assoziierte Funktionen aufgerufen werden, die die entsprechenden Daten zurückgeben.

3. Zuletzt können SUSI und Anwendung durch Ereignisse kommunizieren; es ist möglich, zwischen Situationen in der Dialogbeschreibung und der virtuellen Situation „Anwendung" Trigger auszutauschen.

Realisierung der Schnittstelle zur Präsentationsschicht

Die Präsentationsschicht stellt — zunächst — zwei Kategorien von Elementen zur Kommunikation mit der Benutzerin bzw. dem Benutzer bereit:

1. Wesentlicher Teil ist das eigentliche Anwendungsfenster. Hier erwartet das System Eingaben und veranlaßt entsprechende Aktionen. Bei dem im Rahmen des Projekts realisierten Dateimanager ist dies das Verzeichnisfenster. Anwendungsfenster kann es auch mehrere geben.

2. Für Eingaben, die den Ablauf des Dialogs steuern, werden verschiedene Fenster (Dialogboxen) benötigt, mit denen z.B. Fehlermeldungen, Bestätigungsknöpfe (*OK–Buttons*) o.ä. realisiert sind. Diese werden aufgrund von externen Relationen in der SUSI–Spezifikation aktiviert.

Die Dialogboxen werden dabei durch entsprechende Funktionsaufrufe geöffnet, die im Deklarationsteil der Spezifikation entsprechenden Relationen zugeordnet sind. Die Integration in die Gesamtbeschreibung kann durch direkten Funktionsaufruf erfolgen; das System wartet in diesem Fall auf eine Eingabe und fährt dann mit der Ableitung fort (synchrone Eingabe). In einigen Fällen kann die Dialogbox jedoch auch in einer eigenen Situation aufgerufen werden; die Ableitung wird dann unmittelbar fortgesetzt, ohne auf eine Eingabe zu warten (asynchrone Eingabe). Dies kann beispielsweise bei Fehlermeldungen sinnvoll sein.

Die Kommunikation der Situationen mit den Fenstern der Anwendung kann auf zwei verschiedene Arten stattfinden:

1. Durch *Streams*. Die Kommandos (Trigger oder auswertbare Ausdrücke) werden an einen Stream übertragen und vom Kommunikationspartner gelesen. Dies ermöglicht es, Präsentationsschicht und Dialogsteuerung als verschiedenene Prozesse zu realisieren.

2. Durch *Funktionsaufrufe*. Auswertbare Ausdrücke werden direkt ausgewertet, Trigger an eine Funktion übergeben, die sie in die Warteschlange des Interpretierers einreiht. Im Gegensatz zur ersten Lösung, bei der es für Oberfläche und Dialogsteuerung verschiedene Hauptschleifen gibt, wird hier nur eine benötigt, die im Bereich der Präsentationsschicht angesiedelt ist.

Trigger von der Schnittstelle zur Präsentationsschicht werden wie Trigger innerhalb der Dialogsteuerung behandelt: sie werden in eine Warteschlange eingereiht. Dazu wird eine Schnittstellenfunktion (`run`) bereitgestellt, die bei Ereignissen an der Oberfläche mit dem Trigger als Argument aufgerufen wird. Auswertbare, den Relationen im `declarations`–Teil zugeordnete Ausdrücke, führen zu Seiteneffekten, indem sie als Funktionsaufrufe im LISP–System evaluiert werden.

Realisierung der Schnittstelle zum Hilfssystem

An der Schnittstelle zum Hilfssystem müssen zwei verschiedenene Dienste zur Verfügung stehen:

- Es müssen Informationen zum momentanen Kontext verfügbar sein.
- Es müssen — integriert in den normalen Ablauf — Möglichkeiten der Darstellung von Hilfstexten gegeben sein.

Zu einzelnen Aktionsregeln muß dabei abfragbar sein, ob sie derzeit ausführbar sind und, wenn sie es nicht sind, was dafür der Grund ist. Dazu können zum einen Trigger übergeben werden; dabei wird dann die Regel gesucht, die im Fall eines normalen Aufrufs aufgerufen werden würde. Zum zweiten kann der Bezeichner einer Regel — mit oder ohne einen Trigger — angegeben werden; die simulierte Ableitung ist dann auf diese Regel beschränkt.

Im Fall einer erfolgreichen Ausführung wird an dieser Stelle der Name der Regel zurückgegeben; schlägt die Ableitung — im ersten Fall *aller* getriggerten, im zweiten der angegebenen Regel — fehl, so hat die Funktion die entsprechende Regel und die Relation, die nicht abgeleitet werden konnte, als Ergebnis.

Genauso können Nebenbedingungen getestet werden. Bei Fehlschlagen der Ableitung wird auch hier die nicht ableitbare Relation zurückgegeben.

Wesentlich ist, daß bei der Simulation von Aufrufen keine Seiteneffekte erzeugt werden. Wird eine entsprechende Relation angetroffen, so wird sie sofort zu *wahr* ausgewertet.

Hilfstexte können entweder direkt erzeugt werden, d.h. deren Darstellung wird nicht explizit durch SUSI gesteuert, oder es können besondere Aktionsregeln für sie definiert werden. In diesem Fall werden prozedurale Relationen benötigt, die für die Erzeugung der entsprechenden Fenster sorgen. Außerdem sind zusätzliche Regeln zur Steuerung der Darstellung erforderlich. Für die Fenster sind auch (wie im Fall der Dialogboxen) eigene Situationstypen bzw. Situationen denkbar. Dies ist abhängig von der jeweiligen Implementierung.

5.2 Prototyp und Beispielsitzung

Als eine erste prototypische Anwendung für SUSI wurde im Rahmen der Arbeit [Winter 1994] (Oberflächenobjekte und Hilfstexte) und der vorliegenden Arbeit (SUSI–Dialogspezifikation) ein Dateimanager implementiert. Dieser entspricht allgemein gebräuchlichen Dateimanagern mit Verzeichnisfenster, Pull–Down–Menüs und Fenstern für Abfragen und Fehlermeldungen, erweitert durch kontextabhängige Hilfe.

5.2.1 Funktionalität

In den Dateimanager, der im Rahmen dieser Arbeit implementiert wurde, sind folgende Funktionen realisiert:

- Ausführen entsprechender Befehle bei Auswahl von Pull–Down–Menüs. Dies kann im einzelnen sein:
  - Umbenennen von Dateien
  - Verschieben und Kopieren von Dateien

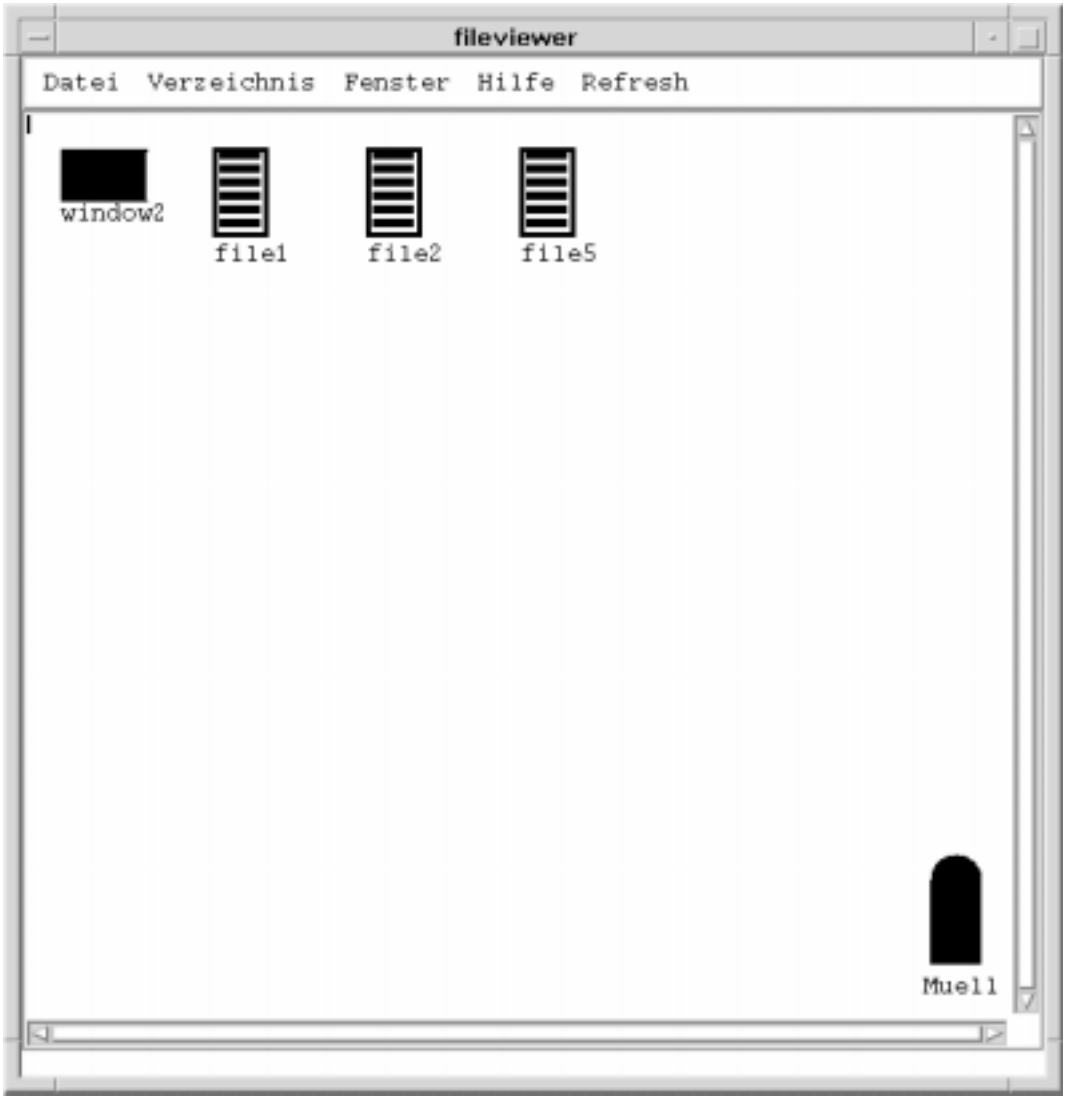

Abbildung 5.2: *Verzeichnisfenster des Dateimanagers*

  - Löschen von Dateien
  - Erstellen von Verzeichnissen
  - Verzeichniswechsel in beliebiges Verzeichnis, in nächsthöheres Verzeichnis, in Heimverzeichnis
  - Öffnen eines Terminalfensters
  - Selektieren/deselektieren von Symbolen

- Kopieren oder Verschieben von Dateien durch Ziehen mit der Maus, entweder auf ein anderes Fenster oder auf ein Ordner–Symbol
- Löschen von Dateien durch Ziehen auf ein Papierkorb–Symbol
- Schrittweises Wechseln von Verzeichnissen durch Doppelklick auf Ordner–Symbole.

Abbildung 5.3: *Dialogfenster zur Anfrage an die Benutzerin/den Benutzer (z.B. Sicherheitsabfrage beim Löschen einer Datei)*

Dabei wird es für bestimmte Operationen verschiedene Wege geben: Eine Datei kann z.B. sowohl durch Verschieben des Symbols als auch durch Wählen der entsprechenden Option der Menüleiste kopiert werden.

Weitere Optionen, wie spezielle Aktionen, die den verschiedenen Dateitypen zugeordnet sind, können leicht hinzugefügt werden. Beispielsweise kann bei Doppelklick auf das Symbol einer Datei mit der Endung „`.tex`“ zur Übersetzung ein Aufruf von LaTeX erfolgen.

5.2.2 Elemente der Oberfläche

Zusätzlich zu den aufgezählten Funktionen sind Abfragen zur Interaktion mit Benutzerin/Benutzer vorgesehen:

- Anfragen wie z.B. Sicherheitsabfragen bei irreversiblen Operationen. Dies geschieht durch den Seiteneffekt einer Relation `queryOK ()`. Die Auswertung der Relation ergibt bei Klick auf „OK“ *wahr* und bei Klick auf „Abbruch“ *falsch* (vgl. Abbildung 5.3).
- Mitteilungen, die bestätigt werden müssen, wie z.B. bei Fehlermeldungen, erfolgen durch einen Seiteneffekt der Relation `queryReport (X)`. Dabei wird eine geeignete Fehlermeldung übergeben (die bei der Auswertung eines vorherigen Prädikats ermittelt worden sein kann). Zurückgegeben wird *wahr*, wenn auf „OK“ geklickt wurde (vgl. Abbildung 5.4).
- Anforderung einer Eingabe wie beispielsweise eines Dateinamens durch einen Seiteneffekt der Relation `queryName (X)`. Wurde ein String eingegeben und auf „OK“ geklickt, so ergibt die Ableitung der Relation *wahr* und `X` wird mit dem eingegebenen String instantiiert. Ansonsten ergibt sie *falsch* (vgl. Abbildung 5.5).

Die eigentlichen Fenster der Anwendung sind die Verzeichnisfenster, in denen Dateien und Unterverzeichnisse durch manipulierbare Symbole dargestellt werden (Abbildung 5.2). Sie sind an eine Situation gekoppelt und werden durch ein prozedurales Fakt `runSit` erzeugt. Dabei soll eine Instanz des Interpretierers als neuer Prozeß gestartet werden.

5.2.3 Trigger zur Darstellung von Ereignissen

Trigger bestimmen intern die Abfolge der Regelausführung und repräsentieren an der Schnittstelle zur Oberfläche Ereignisse, die durch Benutzeraktionen erzeugt werden. Dafür sind die folgenden Möglichkeiten vorgesehen:

Abbildung 5.4: *Dialogfenster zur Mitteilung mit Bestätigung (z.B. Fehlermeldung)*

Abbildung 5.5: *Dialogfenster für eine Eingabeaufforderung (z.B. Dateinamen)*

- `drag (objekt1, objekt2)`: Ziehen des Symbols `objekt1` über das Symbol `objekt2` und loslassen.
- `press (objekt1)`: Drücken des Mausknopfs über einem Symbol (bzw. an einem Punkt der Menüleiste).
- `depress (objekt2)`: Loslassen des Mausknopfs über einem anderen Symbol (bzw. einem Menüpunkt).
- `click (objekt)`: Anklicken (mit sofortigem Loslassen) eines Symbols.
- `doubleClick (objekt)`: Doppelklick auf ein Symbol.

Die Trigger `press` und `depress` stellen dabei eine Zerlegung von `drag` dar; die beiden Möglichkeiten sollten nicht innerhalb desselben Situationstyps verwendet werden. Die Namen der Trigger sind dabei von der tatsächlichen Realisierung der Ereignisse unabhängig; `click` kann z.B. auch durch Druck auf die *linke*, `doubleClick` durch Druck auf die *rechte* Maustaste erzeugt werden.

Beispiel 5.2.1 *Als Beispiel für die Arbeitsweise des Prototypen betrachten wir einen Kopiervorgang innerhalb eines Verzeichnisses: Zunächst wird das Symbol einer Datei selektiert und dann im Menü* `Datei` *der Punkt* `Kopieren` *ausgewählt. Dabei werden zwei Trigger abgesetzt: beim Selektieren des Dateisymbols* `click ('file1')` *und bei der Auswahl des Menüpunktes* `click ('dateiKopieren')`.

Im ersten Schritt wird damit (durch Unifikation des erzeugten Triggers mit der Triggerrelation der Regel) die Aktionsregel

```
fileWaehlen
  trig (click (X)),
  out (backend:extern file (X)),
  not (selected (X))
  -->
  selected (X);
```

aktiviert, die dafür sorgt, daß SUSI *die Datei* `file1` *als ausgewählt* (`selected`) *bekannt gemacht wird (d.h. es wird das Fakt* `selected ('file1')` *etabliert). Im zweiten Schritt wird die Regel*

```
dateiKopieren
  trig (click ('dateiKopieren')), selected (X),
  out (frontend:extern queryName (Y)),
  copy (X, Y)
  -->
  not (selected (X)), fileIcon (Y);
```

angestoßen. Sie ermittelt die ausgewählten Dateien, fragt mit Hilfe der prozeduralen Relation `queryName` *den Namen der zu erstellenden Datei ab (dies ist die in Abbildung 5.6 sichtbare Situation) und führt anschließend die Kopieraktion durch (vorausgesetzt, es wurde ein gültiger Dateiname eingegeben). Die Nachbedingung sorgt abschließend dafür, daß die Datei wieder deselektiert und die neue Datei dargestellt wird.*

Daß die Relation `file` *im ersten Beispiel und die Relation* `queryName` *im zweiten an der Schnittstelle zur Anwendung bzw. zur Präsentationsebene abgeleitet werden, ist durch das vorangestellte* `backend:extern` *bzw.* `frontend:extern` *festgelegt.*

Dieses — recht einfache — Beispiel zeigt, wie die üblichen Interaktionsmechanismen in SUSI spezifiziert werden können. Die eigentliche Stärke bei der Spezifikation liegt nun darin, daß auch komplexe Zusammenhänge, deren Ableitung über das Suchen in einer Datenbasis hinausgehen, beschrieben werden können.

Beispiel 5.2.2 *Ein Beispiel für einen komplexen Zusammenhang ist eine Nebenbedingung, die beschreibt, ob eine Datei verschoben werden kann:*

```
out (backend:extern file (A)) and
out (backend:extern folder (B)) and
currentUser (C) and
out (backend:extern owns (A, C)) and
out (backend:extern owns (B, C))
impl
movable (A, B);
```

Auch hier werden Relationen an der Anwendungsschnittstelle abgeleitet: `file`, `folder` *und* `owns`.

Die Nebenbedingungen müssen dabei nicht auf Horn–Klauseln beschränkt bleiben. Es ist denkbar, darüber später hinauszugehen; diese Möglichkeit wurde bereits im Abschnitt 4.1.3 erörtert.

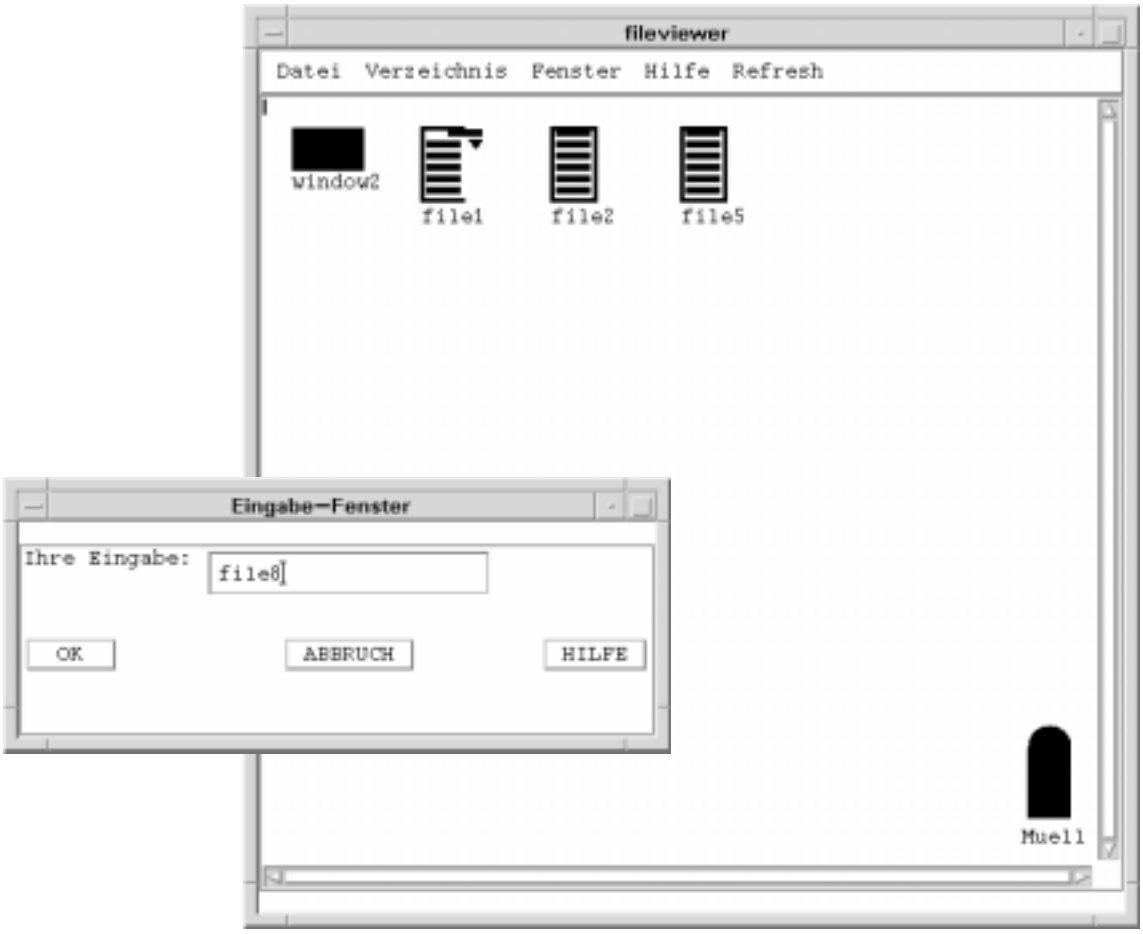

Abbildung 5.6: *Kopieren einer Datei: Ein Dateisymbol wurde selektiert, dann im Menü* `Datei` *der Punkt* `Kopieren` *ausgewählt. In dem kleinen Fenster ist der Name der neuen Datei einzugeben.*

5.2.4 Hilfe

Das zur gleichen Zeit realisierte Hilfssystem ([Winter 1994]) umfaßt sowohl kontextabhängige Hilfe, durch die beispielsweise momentan durchführbare Aktionen erklärt bzw. — wenn sie nicht ausführbar sind — der Grund dafür angegeben wird, als auch statische Hilfe, die z.B. das System als Ganzes beschreibt. Die kontextabhängige Hilfe greift dabei auf das in der SUSI–Spezifikation repräsentierte Wissen zu, indem die Anwendbarkeit von Aktionsregeln durch Auswertung ihrer Vorbedingung ermittelt bzw. Anfragen an die Wissensbasis — einschließlich des in Nebenbedingungen dargestellten Wissens — gerichtet werden. Für diese Ableitungen werden spezielle Funktionen bereitgestellt. Es können so auch notwendige Bedingungen zur Ausführung bestimmter Aktionen (d.h. Relationen) festgestellt werden, die im momentanen Kontext nicht erfüllt sind und damit die Ausführung der Aktion verhindern.

In den Abbildungen 5.7 (kontextabhängige Hilfe) bzw. 5.8 (statische Hilfe) sind dafür Beispiele dargestellt. Ansonsten wird auf die Arbeit [Winter 1994] verwiesen, in der das Thema der (kontextabhängigen) Hilfsfunktionen behandelt wird.

Abbildung 5.7: *Beispiel für kontextabhängige Hilfe*

5.3 Vorgehen bei der Formalisierung

Dieser Abschnitt beschäftigt sich mit der Vorgehensweise bei der Formalisierung der Dialogschnittstelle in SUSI. Es wird dabei davon ausgegangen, daß die Anwendung bereits vorhanden ist und die Oberflächenobjekte bereits in geeigneter Weise realisiert sind.

Soll nun die Dialogkomponente für eine Situation in SUSI implementiert werden, so sind folgende Schritte auszuführen:

1. Die Schnittstellen zu den anderen Komponenten sind festzulegen. Dabei sind bei der Anwendung und bei der Präsentationsschicht verschiedene Dinge zu beachten:

   - Für die Anwendung sind Schnittstellen (in Form von Funktionen) festzulegen, über die semantische Aktionen angestoßen und notwendige Informationen gelesen werden können. Der logische Wert (*wahr* oder *falsch*) wird dabei i.d.R. vom erfolgreichen

Abbildung 5.8: *Beispiel für statische Hilfe*

Ausführen des Seiteneffekts abhängen. Sollen Werte zurückgegeben werden (d.h. Instantiierungen von Variablen erfolgen), so muß bei den Funktionen auf das richtige Format der zurückzugebenden Listen geachtet werden (siehe Abschnitt 5.1).

- Für die Präsentationsebene sind Funktionen festzulegen, die die Darstellung entsprechend der Anforderungen beeinflussen. Diese werden normalerweise dann durch Seiteneffekte der Relationen in der Nachbedingung — abhängig davon, ob diese negiert sind oder nicht — aktiviert. Außerdem müssen dort Trigger festgelegt sein, die entsprechend der Benutzeraktionen zur Steuerung des Dialogs abgesetzt werden.

2. Im `declarations`–Teil werden Relationen deklariert, bei deren Ableitung die Schnittstellenfunktionen aktiviert werden. Die Funktionsaufrufe werden ebenfalls in diesem Teil festgelegt.

3. Aktionen (in Form von Aktionsregeln) sind festzulegen, die durch die vorhandenen Trigger aktiviert werden. Dies geschieht im Abschnitt `actionrules` der Spezifikation.

4. Zuletzt sind Nebenbedingungen (im Abschnitt `constraints`) und ggf. vorher festzulegende Fakten (im Abschnitt `facts`) — getrennt in globale Fakten eines Situationstyps und jeweils in Fakten der einzelnen Situationen — zu spezifizieren.

Zumeist wird sich eine Spezifikation nicht auf eine Situation beschränken, sondern es werden mehrere Situationen — und Situationstypen — vorhanden sein. Dann sind die obigen Schritte für jeden einzelnen Situationstyp durchzuführen. Gleichzeitig sind Trigger zur Kommunikation zwischen den verschiedenen Situationen festzulegen.

5.4 Performanz

Ein Kriterium zur ergonomischen Bewertung von Software ist die *Verfügbarkeit* des Systems. Hier wird gefordert, daß die Benutzung nicht durch Störungen (wie Systemabstürze) oder unangemessene (variierende oder zu lange) Antwortzeiten bei der Arbeit behindert wird. Die akzeptable Länge der Antwortzeiten ist dabei von der Komplexität der Aufgabe abhängig

```
cpu time (non-gc) 1560 msec user, 0 msec system
cpu time (gc) 590 msec user, 0 msec system
cpu time (total) 2150 msec user, 0 msec system
real time 2225 msec
space allocation:
89061 cons cells, 0 symbols, 1571960 other bytes
```

Abbildung 5.9: *Ausgabe der* `time`*–Funktion von Allegro Common Lisp*

([Oppermann et al. 1992]). Shneiderman nennt für viele Aufgaben eine Obergrenze von zwei Sekunden; oft wird aber auch eine unmittelbare Reaktion erwartet, etwa beim Drücken einer Taste ([Shneiderman 1987]).

Bei dem vorliegenden Prototyp, der vor allem dazu dienen soll, die Möglichkeiten des Ansatzes der situationsorientierten Beschreibung von Benutzungsschnittstellen im allgemeinen zu untersuchen, wurde nicht in erster Linie auf eine effiziente Implementierung Wert gelegt. Trotzdem sind Aussagen zur Performanz von Interesse.

Zu diesem Zweck wurden einige Messungen anhand des implementierten Systems durchgeführt, die im folgenden dargestellt werden.

Grundlage war die Entwicklungsumgebung Allegro Common Lisp, zusammen mit dem Editor Emacs. Die Messungen fanden auf einem Rechner von Hewlett–Packard, HP 9000 735 statt. Folgende Operationen wurden gemessen:

1. Verschieben von drei Dateien; dabei wurde die Wissensbasis intern durch Fakten festgelegt (erstes Beispiel in Anhang C).
2. Verschieben von drei Dateien; dabei wurden die notwendigen Informationen mit Hilfe geeigneter Schnittstellenfunktionen aus dem Dateisystem gelesen (zweites Beispiel in Anhang C).
3. Parsen einer Spezifikation (Beispiel im Anhang B).
4. Verschieben einer Datei durch Ziehen mit der Maus.
5. Auswählen aller Sinnbilder eines Verzeichnisses (drei Dateisymbole und ein Verzeichnissymbol) durch Aktivieren des entsprechenden Punktes im Pull–Down–Menü.
6. Auswählen eines einzelnen Dateisymbols durch Anklicken.
7. Wechsel des Verzeichnisses durch doppeltes Anklicken eines Verzeichnissymbols.

Dabei wurde die vom System zur Verfügung gestellte Funktion `time` verwendet, die die Gesamtzeit, die gesamte CPU–Zeit, die CPU–Zeit ohne Garbage–Collection und die CPU–Zeit für die Garbage–Collection ausgibt (Abbildung 5.9).

Die ermittelten Meßwerte sind in Tabelle 5.1 zusammengestellt.

Die Meßwerte für die reine Rechenzeit variieren nur gering; die in der Tabelle ausgewiesenen Schwankungen sind auf die systembedingte Garbage–Collection zurückzuführen und — bei der verwendeten Entwicklungsumgebung — nicht zu vermeiden.

Der Vergleich zwischen den Ergebnissen der ersten beiden Versuche zeigt, daß ein großer Teil der Rechenzeit für Zugriffe auf das Dateisystem verbraucht wird. Diese Zugriffe wurden teilweise in C implementiert; teilweise wurde auf LISP–Funktionen, wie die zeitaufwendige Funktion `run-shell-command` zurückgegriffen, die eine neue UNIX–Shell startet und das ihr übergebene Kommando ausführt. Hier sind sicherlich noch Verbesserungen möglich.

Von weitaus größerem Interesse sind jedoch die Ergebnisse der Versuche 4–7, bei denen Operationen betrachtet wurden, die in realistischen Szenarien auftreten. Einfache Operationen wie das Auswählen eines Symbols benötigen weniger als eine zehntel Sekunde; komplexere Aktionen wie das Verschieben einer Datei — das die Ableitung der Nebenbedingung `movable` einschließt — werden (ohne Garbage Collection) in rund 1.2 Sekunden durchgeführt.

Tabelle 5.1: *Zusammenfassung der bei den Versuchen ermittelten Meßwerte (Angaben in Millisekunden, GC = Garbage–Collection)*

	Gesamtzeit			CPU Gesamt			CPU ohne GC			CPU nur GC	
	Min	Max	∅	Min	Max	∅	Min	Max	∅	Min	Max
1.	919	1657	1214	800	1540	1100	790	830	810	0	750
2.	2777	3425	3102	720	1380	1006	720	810	752	0	640
3.	1233	2124	1592	1190	2070	1542	1190	1200	1192	0	880
4.	1169	2145	1424	210	1040	378	210	230	216	0	810
5.	343	1245	525	330	1220	512	330	340	334	0	890
6.	93	95	94	90	90	90	90	90	90	0	0
7.	60	64	61	60	60	60	60	60	60	0	0

Daß die Geschwindigkeit im Rahmen der verfolgten Ziele kein Problem darstellt, bestätigt der subjektive Eindruck. Wenn auch die Zeiten bei einigen Operationen offensichtlich länger sind als die der ansonsten verwendeten HP–VUE–Arbeitsumgebung, so erweist sich dies nicht als störend. Die Performanz kann somit als akzeptabel bewertet werden.

6

Zusammenfassung und Ausblick

Das Ziel der vorliegenden Arbeit war es, ein System zur Dialogsteuerung für graphische Benutzungsschnittstellen zu entwerfen und zu realisieren. Ausgegangen wurde dabei von dem Ansatz der situationsorientierten Beschreibung nach Strauß, bei der dynamische Abläufe und Kontexte mit Hilfe von Aktionsregeln — gemäß der Aktionstheorie — und logischen Ausdrücken — Formeln im Sinne der Prädikatenlogik erster Stufe — spezifiziert werden. Eine Strukturierung wird in diesem Konzept durch die Einführung von Situationen vorgenommen, durch die jeweils Objekte der Oberfläche — beispielsweise Fenster — beschrieben werden.

Die explizite logische Beschreibung der Dialogsteuerung kann insbesondere dazu verwendet werden, Benutzerinnen und Benutzer mit kontextorientierter Hilfe zu versorgen, die über die in den meisten bestehenden Systemen angebotenen — i.d.R. statischen, d.h. nicht auf den aktuellen Zustand bezogenen — Möglichkeiten der Benutzerunterstützung hinausgeht. Ansätze dazu werden zur gleichen Zeit im Rahmen einer anderen Arbeit innerhalb des Projekts erarbeitet.

Eine kurze Einführung in die unserem Ansatz zugrundeliegenden Theorien — Aktions– und Situationstheorie — findet sich im Abschnitt 2.2.

Die gängigen Konzepte für graphische Benutzungsschnittstellen wurden in Abschnitt 2.1 zusammengestellt. Weitverbreitet sind heutzutage ereignisbasierte Ansätze zur Dialogsteuerung, die eine gute Möglichkeit bieten, direkte Manipulationen von Objekten an der Oberfläche auf Anwendungsaktionen abzubilden. Da dies in den meisten Fällen durch Tabellen geschieht, bestehen jedoch nur beschränkte Möglichkeiten, Wissen über Zustände und Abhängigkeiten der Oberfläche zu repräsentieren und nutzbar zu machen — dieses Wissen befindet sich im Code der Anwendung, der von außen nicht zugänglich ist.

Eine Erweiterung des Ansatzes stellen die produktionsbasierten Konzepte dar, in denen eine Verlagerung von für die Oberfläche relevantem Anwendungswissen in Richtung der Benutzungsschnittstelle stattfindet, das dort durch Fakten — in den meisten Fällen aussagenlogischen Atomen — repräsentiert wird. Eine zusätzliche Erweiterung ist die Formalisierung dynamischen Wissens in Form von Aktionen und der Bedingungen zu ihrer Ausführung. Bei der Ableitung werden jedoch zumeist lediglich Faktenmengen nach den entsprechenden Elementen durchsucht; komplexere Aussagen können so nicht formalisiert werden. In Kapitel 3 wurden drei bestehende Systeme, die diesem Ansatz folgen, vorgestellt.

Die im Rahmen dieser Arbeit realisierte situationsorientierte Beschreibung ist prinzipiell ebenfalls diesem Bereich zuzuordnen. Hier wird aber das produktionsbasierte Konzept einerseits

bezüglich der Ausdrucksmächtigkeit erweitert, indem eine eingeschränkte Form der Prädikatenlogik (ohne Terme) zur Beschreibung der Zustände und von internen Zusammenhängen in der Dialogsteuerung verwendet wird. Insbesondere die Möglichkeit, Nebenbedingungen als (zur Zeit) Hornklauseln darzustellen und der Unifikationsmechanismus, mit dem Variablen instantiiert werden, gehen über die bisherigen Ansätze hinaus.

Andererseits bietet der situationsorientierte Ansatz größere Flexibilität bei der Erzeugung neuer Objekte. Situationen, die die Einheiten der Modularisierung der Wissensbasis darstellen, können bei Bedarf dynamisch erzeugt werden. Auch dies wird von bestehenden Systemen nicht angeboten; findet eine Modularisierung statt, ist diese statisch; zur Laufzeit können keine neuen Module entstehen.

Im Kapitel 4 wurde dann das Konzept zur Realisierung unseres Ansatzes beschrieben. Insbesondere wurde dabei die Formalisierung der Dialogsteuerung als logische Formeln und Aktionsregeln behandelt. Daraus ergab sich dann die konkrete Form der Beschreibung in Form einer Grammatik als Grundlage für einen Interpretierer. Danach wurde die mögliche Architektur dargestellt, Vergleiche zu den zuvor beschriebenen Realisierungen des produktionsbasierten Ansatzes gezogen und das Vorgehen anhand eines ersten Beispiels illustriert.

Auf Eingaben von Benutzerin/Benutzer (d.h. Operationen an der Oberfläche; z.B. Mausklicks) hin werden im System zur Laufzeit spezielle Fakten (Trigger) erzeugt. Es wird dann versucht, diese mit einer ausgezeichneten Relation jeder Aktionsregeln zu unifizieren. Gelingt dies, so werden die entsprechenden Regeln in die engere Wahl für die Ausführung gezogen.

Anschließend wird für eine — nach Maßgabe einer Auswahlfunktion gewählte — Aktionsregel versucht, die als Konjunktion von Relationen formulierte Vorbedingung abzuleiten. Ist dies nicht möglich, wird zur nächsten Regel übergegangen. Gibt es keine weitere Regel, so wird keine Aktion ausgeführt und der nächste Trigger behandelt.

Ansonsten wird die Regel ausgeführt. Dabei werden Veränderungen in der Faktenbasis der Situation und zusätzlich — durch Seiteneffekte — in der Darstellungsebene und der Anwendung bewirkt.

Im Kapitel 5 wurde zuletzt eine prototypische Implementierung des Ansatzes vorgestellt. Das Einlesen der Spezifikation wird durch einen LL–Parser vorgenommen, der die Datenobjekte zur Weiterverarbeitung durch den Interpretierer erzeugt. Dieser verarbeitet zunächst die Aktionsregeln; logische Ableitungen werden von einem rekursiv arbeitenden Resolutionsbeweiser durchgeführt. Die Kommunikation mit den anderen Komponenten, der Darstellungsschicht und der Anwendung erfolgt zum einen über die genannten Trigger, die Ereignisse repräsentieren, und zum anderen über Seiteneffekte, die bestimmten Relationen innerhalb der Spezifikation zugeordnet werden.

Die einleitend zu der Arbeit aufgezählten Eigenschaften spiegeln sich somit folgendermaßen wider: Information über die Benutzungsschnittstelle und die Anwendung ist in der Beschreibung zusammengeführt und explizit dargestellt. Sind dabei einzelne Fakten nicht unmittelbar in der Faktenmenge enthalten, so sind kann auf sie dennoch durch entsprechende Anfragen unmittelbar zugegriffen werden. Ein erweiterter Zugriff durch probeweise Auswertung der Vorbedingungen von Aktionsregeln und von Nebenbedingungen schaffen die Voraussetzungen für die Bereitstellung kontextsensitiver Hilfe.

Zusammen mit dem gleichzeitig entwickelten System zur kontextorientierten Benutzerunterstützung stellt die vorhandene Implementierung so eine Basis für weitere Untersuchungen dar.

Das Konzept bietet dabei noch Möglichkeiten zur Erweiterung. So beschränkt sich die vorliegende Implementierung bezüglich der logischen Ausdrucksmächtigkeit auf Hornklauseln, mit denen Ableitungen einigermaßen effizient durchgeführt werden können. Im Abschnitt 4.1.3 wurde bereits über eine Erweiterung bezüglich der Ausdrucksmächtigkeit nachgedacht; weitere Überlegungen dazu wären sicherlich von Interesse; beispielsweise um die Möglichkeit von Integritätsüberprüfungen der Wissensbasis zu bieten. Dabei müssen jedoch die Ausführungszeiten im Auge behalten werden. Auch die Möglichkeit einer Vorausberechnung von Vorbedingungen (ähnlich wie in PPS), um Wissen über ausführbare Benutzeraktionen zu erhalten, ist denkbar. Hier muß zuvor untersucht werden, wie die Vorbedingungen von Aktionsregeln (bzw. ihre Trigger) auf Bildschirmobjekte abgebildet werden können, um zu ermitteln, ob sie aktivierbar sind.

Die Mengen der Aktionsregeln besitzen in der gegenwärtigen Konzeption keine weitere Strukturierung. Bis zu einem bestimmten Grad an Komplexität der zu beschreibenden Objekte ist das sicherlich unproblematisch: die Regelmenge des Prototypen, der die wesentliche Funktionalität eines Dateimanagers bietet, ist noch ohne weiteres zu überschauen. Es ist jedoch nicht auszuschließen, daß bei anderen Beispielen dieser Grad überschritten wird; es können dann unter Umständen größere, unübersichtliche Regelmengen entstehen. In diesem Fall wäre über Möglichkeiten weiterer Strukturierung nachzudenken.

Einleitend zu der Arbeit wurde auf die Selbstbeschreibungsfähigkeit von Software–Systemen als Kriterium der Software–Ergonomie eingegangen. Das Projekt SUSI (und damit auch diese Arbeit) stellt einen Versuch dar, diese Selbstbeschreibungsfähigkeit zu erhöhen. Durch eine adäquate Darstellung der Abläufe in einem System werden zum einen Benutzerinnen und Benutzer geschult (was natürlich keinesfalls eine Schulung in Form von Lehrgängen o.ä. ersetzen darf), sowie darin unterstützt, die Arbeitsweise von Systemen — hier insbesondere die Reaktionen auf Eingaben — nachvollziehen zu können. Insofern ist es das Ziel, einen Beitrag zu einem besseren Verständnis von Software–Systemen zu leisten. Entsprechende Untersuchungen, die im weiteren Verlauf des Projektes durchgeführt werden sollen, können dazu beitragen, sowohl den gesamten Ansatz im Hinblick auf (beispielsweise) die Repräsentationsformalismen zu evaluieren als auch geeignete Vorgehensweisen der konkreten Beschreibung in der Spezifikationssprache von SUSI zu finden.

A

Die Grammatik der Beschreibungssprache

Im folgenden ist die Grammatik der Beschreibungssprache von SUSI dargestellt. Zur Implementierung eines LL–Parsers wurden vor der Implementierung Linksrekursionen eliminiert (nach einem Algorithmus in [Waite, Goos 1984]).

specification	::=	"situationtypes" specifications "end"
specifications	::=	specForOneSitTyp specifications |
		ϵ
specForOneSitTyp	::=	situationTypId actionRules constraints facts "end"
actionRules	::=	"actionrules" setOfActionRules "end"
setOfActionRules	::=	singleActionRule setOfActionRules |
		ϵ
singleActionRule	::=	actionRuleId ":" precond "−− >" postcond ";"
precond	::=	preconds
preconds	::=	inTrigger preconds1
preconds1	::=	relation "," preconds1 |
		relation |
		ϵ
postcond	::=	postconds
postconds	::=	literalOrOutTrigger "," postconds |
		literalOrOutTrigger |
		ϵ
actionRuleId	::=	symbol
literalOrOutTrigger	::=	literal |
		outTrigger
constraints	::=	"constraints" setOfConstraints "end"
setOfConstraints	::=	singleConstraint setOfConstraints |
		ϵ
singleConstraint	::=	constraintImpl ";"
constraintImpl	::=	constraintImpl "impl" constraintOr |
		constraintOr
constraintOr	::=	constraintOr "or" constraintAnd |
		constraintAnd

constraintAnd	::=	constraintAnd “and” constraintNot |
		constraintNot
constraintNot	::=	relation |
		“not” “(” constraintImpl “)” |
		“(” constraintImpl “)”
facts	::=	“facts” setOfTypeFacts setOfSitFacts “end”
setOfTypeFacts	::=	“type” factsForOneSit “end”
setOfSitFacts	::=	setOfFactsForOneSit setOfSitFacts |
		ϵ
setOfFactsForOneSit	::=	situationId factsForOneSit “endsit”
factsForOneSit	::=	singleFact factsForOneSit |
		ϵ
singleFact	::=	atomicFormula “;”
relation	::=	literal |
		out
inTrigger	::=	“trig” “(” atomicFormula “)”
outTrigger	::=	“trig” “(” situationId “:” situationTypId atomicFormula “)”
out	::=	“out” “(” situationId “:” situationTypId literal “)”
situationId	::=	symbol
situationTypId	::=	symbol
literal	::=	atomicFormula |
		“not” atomicFormula
atomicFormula	::=	symbol “(” args “)”
args	::=	symbol |
		symbol “,” args |
		ϵ
symbol	::=	char |
		char symbol
char	::=	“A” |“B” |“C” |“D” |“E” |“F” |“G” |“H” |“I” |
		“J” |“K” |“L” |“M” |“N” |“O” |“P” |“Q” |“R” |
		“S” |“T” |“U” |“V” |“W” |“X” |“Y” |“Z” |
		“a” |“b” |“c” |“d” |“e” |“f” |“g” |“h” |“i” |
		“j” |“k” |“l” |“m” |“n” |“o” |“p” |“q” |“r” |
		“s” |“t” |“u” |“v” |“w” |“x” |“y” |“z”

B

Beispiel für eine Dialogspezifikation

Dieser Abschnitt enthält ein lauffähiges Beispiel für eine mit SUSI spezifizierte Dialogkomponente für einen einfachen Dateimanager mit Symbolen für Verzeichnisse und Dateien und mit Pull–Down–Menüs.

Die Beschreibung zerfällt in zwei Teile: der erste Teil definiert die Schnittstelle zu Oberfläche und Anwendung, d.h. die Bedeutung prozeduraler Relationen. Der jeweiligen Relation ist ein LISP–Ausdruck zugeordnet, der bei der Ableitung der Relation ausgewertet wird.

Eine Darstellung der Arbeitsweise mit dazugehörigen Abbildungen findet sich im Abschnitt 5.2.

```
declarations
   move () : semantic "(clim-user::rename-files \"#1\" \"#2\")";
   copy () : semantic "(clim-user::copy-file \"#1\" \"#2\")";
   rm () : semantic "(clim-user::delete-file \"#1\")";
   folder () : semantic "(clim-user::FOLDER \"(folder #1)\")";
   file () : semantic "(clim-user::FILE \"(file #1)\")";
   owns () : semantic "(clim-user::OWNS \"(owns #1 #2)\")";
   queryName () : semantic "(clim-user::QUERY-NAME \"(queryName #1)\")";
   queryReport () : semantic "(clim-user::QUERY-REPORT \"(queryReport #1)\")";
   queryOK () : semantic "(clim-user::QUERY-OK \"(queryOK nil)\")";
   create () : semantic "(open \"#1\" :if-does-not-exist :create)";
   mkdir () : semantic "(excl:run-shell-command \"mkdir #1\")";
   chdir () : semantic "(clim-user::CHANGEDIR \"#1\")";
   chdirHome () : semantic "(clim-user::CHANGEDIRHOME)";
   newSit () : semantic "(clim-user::init-window)"
   term () : semantic "(excl:run-shell-command \"hpterm&\")";

   fileIcon () : presentation
     "(clim-user::make-file-icon (clim-user::stringToSymbol \"#1\"))" :
     "(clim-user::remove-icon (clim-user::stringToSymbol \"#1\"))";
   folder-icon () : presentation
     "(clim-user::make-folder-icon (clim-user::stringToSymbol \"#1\"))" :
     "(clim-user::remove-icon (clim-user::stringToSymbol \"#1\"))";
   selected () : presentation
     "(clim-user::select-icon (clim-user::stringToSymbol \"#1\"))" :
     "(clim-user::unselect-icon (clim-user::stringToSymbol \"#1\"))";
end
```

Der zweite Teil ist die eigentliche Spezifikation. Hier sind die Aktionen und ihre logischen Bedingungen festgelegt. Dabei wird teilweise auf die „virtuellen“ Situationen Anwendung (`backend`) und Präsentationsschicht (`frontend`) zugegriffen.

```
situationtypes
  fileviewer
    actionrules
      start
        trig (init ()), out (frontend:extern newSit ('window1'))
        -->
        ;
      dateiErstellen
        trig (click ('dateiErstellen')), out (frontend:extern queryName (X)),
        create (X)
        -->
        fileIcon (X);
      dateiUmbenennen
        trig (click ('dateiUmbenennen')), selected (X),
        out (frontend:extern queryName (Y)),
        rename (X, Y)
        -->
        not (selected (X)), not (fileIcon (X)), fileIcon (Y);
      dateiKopieren
        trig (click ('dateiKopieren')), selected (X),
        out (frontend:extern queryName (Y)),
        copy (X, Y)
        -->
        not (selected (X)), fileIcon (Y);
      dateiInPapierkorb
        trig (click ('dateiInPapierKorb')), selected (X),
        out (frontend:extern queryOK ()),
        rm (X)
        -->
        not (selected (X)), not (fileIcon (X));
      exit
        trig (click ('exit')), quit ()
        -->
        ;
      verzErstellen
        trig (click ('verzErstellen')),
        out (frontend:extern queryName (X)),
        mkdir (X)
        -->
        folder-icon (X);
      verzWechsel
        trig (click ('verzWechsel')),
        out (frontend:extern queryName (X)),
        chdir (X)
        -->
        ;
      verzHeim
        trig (click ('verzHeimverzeichnis')),
        chdirHome ()
        -->
        ;
```

```
verzHoch
  trig (click ('verzNaechstHoeheresVerzeichnis')),
  chdir ('..')
  -->
  ;
openTerminal
  trig (click ('verzTerminal'),
  term ()
  -->
  ;
fenstJedeWaehlen
  trig (click ('fenstJedeWaehlen'))
  -->
  trig (window1:fileviewer fileJedeWaehlen ()),
  trig (window1:fileviewer folderJedeWaehlen ());
fenstFileJedeWaehlen
  trig (fileJedeWaehlen ()),
  out (backend:extern file (X)),
  not (selected (X))
  -->
  selected (X);
fenstFolderJedeWaehlen
  trig (folderJedeWaehlen ()),
  out (backend:extern folder (X)),
  not (selected (X))
  -->
  selected (X);
fenstKeineWaehlen
  trig (click ('fenstKeineWaehlen')),
  selected (X)
  -->
  not (selected (X));
fileWaehlen
  trig (click (X)),
  out (backend:extern file (X)),
  not (selected (X))
  -->
  selected (X);
folderWaehlen
  trig (click (X)),
  out (backend:extern folder (X)),
  not (selected (X))
  -->
  selected (X);
abWaehlen
  trig (click (X)), selected (X)
  -->
  not (selected (X));
dragInPapierkorb
  trig (drag (X, 'papierKorb'))
  out (frontend:extern queryOK ()),
  rm (X)
  -->
  not (fileIcon (X));
```

```
      dragNotInPapierKorb
        trig (drag (X, 'papierKorb'))
        -->
        fileIcon (X);
      dragInVerzeichnisAndCopy
        trig (drag (Y, X)), not (movable (Y, X)), copy (Y, X)
        -->
        fileIcon (Y);
      dragInVerzeichnisAndMove
        trig (drag (Y, X)), movable (Y, X), move (Y, X)
        -->
        ;
      openFolderByDoubleClick
        trig (doubleClick (X)), out (backend:extern folder (X)), chdir (X)
        -->
        ;
    end

    constraints
      out (backend:extern file (A)) and out (backend:extern folder (B)) and
      currentUser (C) and out (backend:extern owns (A, C)) and
      out (backend:extern owns (B, C))
      impl
      movable (A, B);
    end

    facts
      type
        currentUser ('stefan');
      end
      window1
      endsit
    end
  end
end
```

C

Weitere Beispiele für Spezifikationen

In diesem Abschnitt sind zwei weitere Beispiele enthalten, die für die Zeitmessungen im Abschnitt 5.4 herangezogen wurden. Sie unterscheiden sich darin, daß der Inhalt des bearbeiteten Verzeichnisses und Eigenschaften der Dateien im ersten Fall als Fakten in der Spezifikation festgelegt sind, wogegen sie im zweiten Fall von der Anwendungsschnittstelle (der Schnittstelle zum Dateisystem) zur Laufzeit gelesen werden.

Beispiel ohne Zugriff auf das Dateisystem

```
declarations
  move () : semantic "(clim-user::rename-files \"#1\" \"#2\")";
  copy () : semantic "(clim-user::copy-file \"#1\" \"#2\")";
end

situationtypes
  fileviewer
    actionrules
      start
        trig (init ()), file (X)
        -->
        selected (X),
        trig (window1:fileviewer drag ('window1', 'window2'));
      ziehen
        trig (drag ('window1', Ziel))
        -->
        trig (window1:fileviewer startCopy (Ziel)),
        trig (window1:fileviewer startMove (Ziel));
      kopieren
        trig (startCopy (Ziel)), currentUser (Z), selected (X),
        not (movable (X, Ziel)), copy (X, Ziel)
        -->
        trig (Ziel:fileviewer display (X));
```

```
      verschieben
        trig (startMove (Ziel)), selected (X), currentUser (Z),
        movable (X, Ziel), move (X, Ziel)
        -->
        trig (Ziel:fileviewer display (X)),
        trig (window1:fileviewer remove (X));
      darstellen
        trig (display (Z))
        -->
        file (Z), fileIcon (Z);
      entfernen
        trig (remove (Z))
        -->
        not (file (Z)), not (fileIcon (Z));
    end

    constraints
      file (X) and folder (Y) and currentUser (Z) and
      owns (X, Z) and owns (Y, Z)
      impl
      movable (X, Y);
    end

    facts
      type
        currentUser ('stefan');
      end
      window1
        folder ('window1');
        folder ('window2');
        file ('file1');
        file ('file2');
        file ('file5');
        folderIcon ('window1');
        folderIcon ('window2');
        fileIcon ('file1');
        fileIcon ('file2');
        fileIcon ('file5');
        owns ('window1', 'stefan');
        owns ('window2', 'stefan');
        owns ('file1', 'stefan');
        owns ('file2', 'stefan');
        owns ('file5', 'stefan');
      endsit
      window2
      endsit
    end
  end
end
```

Beispiel mit Zugriff auf das Dateisystem

```
declarations
  move () : semantic "(clim-user::rename-files \"#1\" \"#2\")";
  copy () : semantic "(clim-user::copy-file \"#1\" \"#2\")";
  folder () : semantic "(clim-user::FOLDER \"(folder #1)\")";
  file () : semantic "(clim-user::FILE \"(file #1)\")";
  owns () : semantic "(clim-user::OWNS \"(owns #1 #2)\")";
end


situationtypes
  fileviewer
    actionrules
      start
        trig (init ()),
        out (backend:extern file (X)),
        -->
        selected (X),
        trig (window1:fileviewer drag ('window1', 'window2'));
      ziehen
        trig (drag ('window1', Ziel))
        -->
        trig (window1:fileviewer startCopy (Ziel)),
        trig (window1:fileviewer startMove (Ziel));
      kopieren
        trig (startCopy (Ziel)), selected (X), currentUser (Z),
        not (movable (X, Ziel)),
        copy (X, Ziel)
        -->
        trig (Ziel:fileviewer display (X));
      verschieben
        trig (startMove (Ziel)), selected (X), currentUser (Z),
        movable (X, Ziel),
        move (X, Ziel)
        -->
        trig (Ziel:fileviewer display (X)),
        trig (window1:fileviewer remove (X));
      darstellen
        trig (display (Z))
        -->
        fileIcon (Z);
      entfernen
        trig (remove (Z))
        -->
        not (fileIcon (Z));
    end


    constraints
      out (backend:extern file (X)) and out (backend:extern folder (Y)) and
      currentUser (Z) and out (backend:extern owns (X, Z)) and
      out (backend:extern owns (Y, Z))
      impl
      movable (X, Y);
    end
```

```
    facts
      type
        currentUser ('stefan');
      end
      window1
      endsit
      window2
      endsit
    end
  end
end
```

D

Protokoll einer Ableitung

Dieser Abschnitt enthält das Protokoll einer Ableitung des ersten Beispiels im Anhang C. Um die Funktionsweise der Nebenbedingung `movable` zu verdeutlichen, wurde im Abschnitt `facts` das Fakt `owns (file5, stefan)` durch `owns (file5, niklas)` ersetzt. Es ist zu beachten, daß der erste Aufruf von `movable` negiert erfolgt.

Um die Lesbarkeit zu erhöhen, wurden an einigen Stellen Einrückungen vorgenommen bzw. Leerzeilen eingefügt.

```
Regel angestossen durch Trigger (init nil) in Situation window1:fileviewer.
Liste aktueller Trigger: nil

Praedikat (file X) an Beweiser uebergeben.
 Anfrage: ((file X) nil)
 Regel: ((file file1)) <-- nil
 Anfrage: ((file X) nil)
 Regel: ((file file2)) <-- nil
 Anfrage: ((file X) nil)
 Regel: ((file file5)) <-- nil
Ableitung von ((file X) nil) erfolgreich.
Ergebnis: ((file X) (((X file5)) ((X file2)) ((X file1))))
 Neues Fakt: (selected file5)
 Neues Fakt: (selected file2)
 Neues Fakt: (selected file1)

Regel angestossen durch Trigger (drag window1 window2)
in Situation window1:fileviewer.
Liste aktueller Trigger: nil

Regel angestossen durch Trigger (startCopy Ziel) in
Situation window1:fileviewer.
Liste aktueller Trigger: ((startMove Ziel))

Praedikat (currentUser Z) an Beweiser uebergeben.
 Anfrage: ((currentUser Z) nil)
 Regel: ((currentUser stefan)) <-- nil
Ableitung von ((currentUser Z) nil) erfolgreich.
Ergebnis: ((currentUser Z) (((Z stefan))))
```

```
Praedikat (selected X) an Beweiser uebergeben.
 Anfrage: ((selected X) nil)
 Regel: ((selected file1)) <-- nil
 Anfrage: ((selected X) nil)
 Regel: ((selected file2)) <-- nil
 Anfrage: ((selected X) nil)
 Regel: ((selected file5)) <-- nil
Ableitung von ((selected X) nil) erfolgreich.
Ergebnis: ((selected X) (((X file5)) ((X file2)) ((X file1))))


Praedikat (movable X Ziel) an Beweiser uebergeben.
 Anfrage: ((movable X Ziel) nil)
 Regel: ((movable X Y)) <-- ((file X) (folder Y) (currentUser Z) (owns Y Z)
                             (owns X Z))
 Anfrage: ((file X) nil)
 Regel: ((file file1)) <-- nil
 Anfrage: ((file X) nil)
 Regel: ((file file2)) <-- nil
 Anfrage: ((file X) nil)
 Regel: ((file file5)) <-- nil
((file X) (((X file5)) ((X file2)) ((X file1)))) abgeleitet.
 Anfrage: ((folder Y) nil)
 Regel: ((folder window1)) <-- nil
 Anfrage: ((folder Y) nil)
 Regel: ((folder window2)) <-- nil
((folder Y) (((Y window2)) ((Y window1)))) abgeleitet.
 Anfrage: ((currentUser Z) nil)
 Regel: ((currentUser stefan)) <-- nil
((currentUser Z) (((Z stefan)))) abgeleitet.
 Anfrage: ((owns Y Z) nil)
 Regel: ((owns window1 stefan)) <-- nil
 Anfrage: ((owns Y Z) nil)
 Regel: ((owns window2 stefan)) <-- nil
 Anfrage: ((owns Y Z) nil)
 Regel: ((owns file1 stefan)) <-- nil
 Anfrage: ((owns Y Z) nil)
 Regel: ((owns file2 stefan)) <-- nil
 Anfrage: ((owns Y Z) nil)
 Regel: ((owns file5 niklas)) <-- nil
((owns Y Z)
 (((Y file5) (Z niklas)) ((Y file2) (Z stefan)) ((Y file1) (Z stefan))
  ((Y window2) (Z stefan)) ((Y window1) (Z stefan)))) abgeleitet.
 Anfrage: ((owns X Z) nil)
 Regel: ((owns file5 niklas)) <-- nil
 Anfrage: ((owns X Z) nil)
 Regel: ((owns file2 stefan)) <-- nil
 Anfrage: ((owns X Z) nil)
 Regel: ((owns file1 stefan)) <-- nil
 Anfrage: ((owns X Z) nil)
 Regel: ((owns window2 stefan)) <-- nil
 Anfrage: ((owns X Z) nil)
 Regel: ((owns window1 stefan)) <-- nil
((owns X Z)
 (((X window1) (Z stefan)) ((X window2) (Z stefan))
  ((X file1) (Z stefan)) ((X file2) (Z stefan)) ((X file5) (Z niklas))))
abgeleitet.
```

```
Ableitung von ((movable X Ziel) nil) erfolgreich.
Ergebnis: ((movable X Ziel)
           (((Ziel window2) (X file1)) ((Ziel window1) (X file1))
            ((Ziel window2) (X file2)) ((Ziel window1) (X file2))))

Von externem Praedikat (copy X Ziel) Befehl(e):
 (clim-user::copy-file "file5" "window2")
an Schnittstelle uebergeben.


Regel angestossen durch Trigger (startMove Ziel)
in Situation window1:fileviewer.
Liste aktueller Trigger: ((display X))

Praedikat (selected X) an Beweiser uebergeben.
 Anfrage: ((selected X) nil)
 Regel: ((selected file1)) <-- nil
 Anfrage: ((selected X) nil)
 Regel: ((selected file2)) <-- nil
 Anfrage: ((selected X) nil)
 Regel: ((selected file5)) <-- nil
Ableitung von ((selected X) nil) erfolgreich.
Ergebnis: ((selected X) (((X file5)) ((X file2)) ((X file1))))

Praedikat (currentUser Z) an Beweiser uebergeben.
 Anfrage: ((currentUser Z) nil)
 Regel: ((currentUser stefan)) <-- nil
Ableitung von ((currentUser Z) nil) erfolgreich.
Ergebnis: ((currentUser Z) (((Z stefan))))

Praedikat (movable X Ziel) an Beweiser uebergeben.
 Anfrage: ((movable X Ziel) nil)
 Regel: ((movable X Y)) <-- ((file X) (folder Y) (currentUser Z) (owns Y Z)
                             (owns X Z))
 Anfrage: ((file X) nil)
 Regel: ((file file1)) <-- nil
 Anfrage: ((file X) nil)
 Regel: ((file file2)) <-- nil
 Anfrage: ((file X) nil)
 Regel: ((file file5)) <-- nil
((file X) (((X file5)) ((X file2)) ((X file1)))) abgeleitet.
 Anfrage: ((folder Y) nil)
 Regel: ((folder window1)) <-- nil
 Anfrage: ((folder Y) nil)
 Regel: ((folder window2)) <-- nil
((folder Y) (((Y window2)) ((Y window1)))) abgeleitet.
 Anfrage: ((currentUser Z) nil)
 Regel: ((currentUser stefan)) <-- nil
((currentUser Z) (((Z stefan)))) abgeleitet.
 Anfrage: ((owns Y Z) nil)
 Regel: ((owns window1 stefan)) <-- nil
 Anfrage: ((owns Y Z) nil)
 Regel: ((owns window2 stefan)) <-- nil
 Anfrage: ((owns Y Z) nil)
 Regel: ((owns file1 stefan)) <-- nil
 Anfrage: ((owns Y Z) nil)
 Regel: ((owns file2 stefan)) <-- nil
```

```
 Anfrage: ((owns Y Z) nil)
 Regel: ((owns file5 niklas)) <-- nil
((owns Y Z)
 (((Y file5) (Z niklas)) ((Y file2) (Z stefan)) ((Y file1) (Z stefan))
  ((Y window2) (Z stefan)) ((Y window1) (Z stefan)))) abgeleitet.
 Anfrage: ((owns X Z) nil)
 Regel: ((owns file5 niklas)) <-- nil
 Anfrage: ((owns X Z) nil)
 Regel: ((owns file2 stefan)) <-- nil
 Anfrage: ((owns X Z) nil)
 Regel: ((owns file1 stefan)) <-- nil
 Anfrage: ((owns X Z) nil)
 Regel: ((owns window2 stefan)) <-- nil
 Anfrage: ((owns X Z) nil)
 Regel: ((owns window1 stefan)) <-- nil
((owns X Z)
 (((X window1) (Z stefan)) ((X window2) (Z stefan))
  ((X file1) (Z stefan)) ((X file2) (Z stefan)) ((X file5) (Z niklas))))
abgeleitet.
Ableitung von ((movable X Ziel) nil) erfolgreich.
Ergebnis: ((movable X Ziel)
           (((Ziel window2) (X file1)) ((Ziel window1) (X file1))
            ((Ziel window2) (X file2)) ((Ziel window1) (X file2))))

Von externem Praedikat (move X Ziel) Befehl(e):
 (clim-user::rename-files "file1" "window2")
 (clim-user::rename-files "file2" "window2")
an Schnittstelle uebergeben.

Regel angestossen durch Trigger (display X) in Situation window2:fileviewer.
Liste aktueller Trigger: ((display X) (remove X))
 Neues Fakt: (file file5)
 Neues Fakt: (fileIcon file5)

Regel angestossen durch Trigger (display X) in Situation window2:fileviewer.
Liste aktueller Trigger: ((remove X))
 Neues Fakt: (file file1)
 Neues Fakt: (file file2)
 Neues Fakt: (fileIcon file1)
 Neues Fakt: (fileIcon file2)

Regel angestossen durch Trigger (remove X) in Situation window1:fileviewer.
Liste aktueller Trigger: nil
 Fakt entfernt: (file file1)
 Fakt entfernt: (file file2)
 Fakt entfernt: (fileIcon file1)
 Fakt entfernt: (fileIcon file2)
```

In dem Beispiel werden zunächst die Dateien `file1` und `file2`, die beide den Benutzer `stefan` gehören, und die Datei `file5`, die dem Benutzer `niklas` gehört, selektiert. Anschließend wird letztere (für die `movable` nicht erfüllt ist) kopiert und die beiden ersten (für die `movable` erfüllt ist) verschoben. Den Abschluß bildet die Aktualisierung der Faktenmengen.

Literatur

[Balzert et al. 1988] Helmut Balzert, Heinz U. Hoppe, Reinhard Oppermann, Helmut Peschke, Gabriele Rohr, Norbert A. Streitz (Hrsg.): *Einführung in die Software–Ergonomie*, de Gruyter, 1988

[Bancilhon, Ramakrishnan 1988] François Bancilhon, Raghu Ramakrishnan: *Performance Evaluation of Data Intensive Logic Programs*, in: Jack Minker (Hrsg.): *Foundations of Deductive Databases and Logic Programming*, Morgan Kaufmann Publishers, Los Altos, California, 1988

[Barwise, Perry 1983] Jon Barwise, John Perry: *Situations and Attitudes*, Bradford Books, MIT Press, 1983

[Clocksin, Mellish 1987] William F. Clocksin, Christopher S. Mellish: *Programming in Prolog*, 3. Auflage, Springer–Verlag, 1987

[Devlin 1991] Keith Devlin: *Logic and Information*, Cambridge University Press, 1991

[Foley et al. 1989] James D. Foley, Won Chul Kim, Srdjan Kovačević, Kevin Murray: *Defining Interfaces at a High Level of Abstraction*, in: *IEEE Software*, Vol. 6, No. 1, 1989

[Franz 1992] *Allegro Common Lisp User Guide*, Version 4.1, 2 Bände, Franz Inc., 1992

[Gieskens, Foley 1991] Daniel F. Gieskens, James D. Foley: *Controlling User Interface Objects Through Pre– and Postconditions*, Technical Report GIT–GVU–91–09, Georgia Institute of Technology, Atlanta, Georgia, 1991

[de Graaff 1992] Johannes J. de Graaff: *Context–sensitive Help as an integral Part of a User Interface Design Environment*, Master's thesis, Delft University of Technology, Faculty of Technical Mathematics and Informatics, Department of Information Systems, 1992

[de Graaff et al. 1993] Johannes J. de Graaff, Piyawadee „Noi“ Sukaviriya, Charles A. P. G. van der Mast: *Automatic Generation of Context–Sensitive Textual Help*, Technical Report GIT–GVU–93–11, Georgia Institute of Technology, Atlanta, Georgia, 1993

[Hartson, Hix 1989] H. Rex Hartson, Deborah Hix: *Human–Computer Interface Development: Concepts and Systems for its Management*, in: *ACM Computing Surveys*, Vol. 21, No. 1, 1989

[Hewlett–Packard 1991] *HP C/HP–UX Reference Manual*, 2. Auflage, Hewlett–Packard Company, 1991

[Hill 1986] Ralph D. Hill: *Supporting Concurrency, Communication and Sychronisation in Human–Computer Interaction — The Sassafras UIMS*, in: *ACM Transactions on Graphics*, Vol. 5, No. 3, 1986

[Ilg, Ziegler 1988] Rolf Ilg, Jürgen Ziegler: *Direkte Manipulation*, in: H. Balzert, H. U. Hoppe, R. Oppermann, H. Peschke, G. Rohr, N. A. Strelitz (Hrsg.): *Einführung in die Software–Ergonomie*, de Gruyter, 1988

[Kartha 1993] G. Neelakantan Kartha: *Soundness and Completeness Theorems for Three Formalizations of Action*, in: Ruzena Bajcsy (Hrsg.) *Proceedings of the IJCAI'93 (13^{th} International Joint Conference on Artificial Intelligence)*, Chambéry, France, 1993

[Keene 1989] Sonya E. Keene: *Object–Oriented Programming in Common LISP — A Programmer's Guide to CLOS*, Addison–Wesley, 1989

[Lifschitz 1987] Vladimir Lifschitz: *Formal Theories of Action*, in: Frank M. Brown (Hrsg.): *The Frame Problem in Artificial Intelligence: Proceedings of the 1987 Workshop*, Lawrence, Kansas, 1987

[Maaß 1993] Susanne Maaß: *Software–Ergonomie — Benutzer– und Aufgabenorientierte Systemgestaltung*, in: *Informatik–Spektrum*, Vol. 16, No. 4, 1993

[Menzel, Schmitt 1993] Wolfram Menzel, Peter H. Schmitt: *Formale Systeme*, Vorlesungsskriptum, Institut für Logik, Komplexität und Deduktionssysteme, Fakultät für Informatik, Universität Karlsruhe, 1993

[Myers 1989] Brad A. Myers: *User–Interface Tools: Introduction and Survey*, in: *IEEE Software*, Vol. 6, No. 1, 1989

[Myers 1993] Brad A. Myers: *State of the Art in User Interface Software Tools*, in: H. Rex Hartson, Deborah Hix (Hrsg.): *Advances in Human–Computer Interaction*, Vol. 4, Ablex Publishing Corporation, Norwood, New Jersey, 1993

[Olsen] Dan R. Olsen, Jr.: *Propositional Production Systems for Dialog Description*, Technical Report, Computer Science Department, Brigham Young University, Provo, Utah, ohne Jahresangabe

[Olsen 1992] Dan R. Olsen, Jr.: *User Interface Management Systems: Models and Algorithms*, Morgan Kaufmann Publishers, San Mateo, California, 1992

[Oppermann et al. 1992] Reinhard Oppermann, Bernd Murchner, Harald Reiterer, Manfred Koch: *Softwareergonomische Evaluation — der Leitfaden EVADIS II*, 2. Auflage, de Gruyter, 1992

[Pednault 1986] Edwin P. D. Pednault: *Formulating Multiagent, Dynamic–World Problems in the classical Planning Framework*, in: Michael P. Georgeff, Amy L. Lansky (Hrsg.): *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, Timberline, Oregon, 1986

[Pednault 1989] Edwin P. D. Pednault: *ADL: Exploring the Middle Ground between STRIPS and the Situation Calculus*, in: Ronald J. Brachman, Hector J. Levesque, Raymond Reiter (Hrsg.): *Proceedings of the 1^{st} International Conference on Principles of Knowledge Representation and Reasoning*, Toronto, Ontario, Kanada, 1989

[Puppe 1988] Frank Puppe: *Einführung in Expertensysteme*, Studienreihe Informatik, Springer–Verlag, 1988

[Schmitt 1991] Peter H. Schmitt: *Theorie der Logischen Programmierung*, Vorlesungsskriptum, Institut für Logik, Komplexität und Deduktionssysteme, Fakultät für Informatik, Universität Karlsruhe, 1991

[Shneiderman 1987] Ben Shneiderman: *Designing the User Interface: Strategies for Effective Human–Computer Interaction*, Addison–Wesley Publishing Company, 1987

[Shoham 1986] Yoav Shoham: *What is the Frame Problem?*, in: Michael P. Georgeff, Amy L. Lansky (Hrsg.): *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, Timberline, Oregon, 1986

[Steele 1990] Guy L. Steele, Jr.: *Common LISP — The Language*, 2. Auflage, Digital Press, 1990

[Strauß 1992] Friedrich Strauß: *Situationsorientierte Modellierung für graphische Benutzungsoberflächen*, IIG–Bericht 5/92, Institut für Informatik und Gesellschaft, Universität Freiburg, 1992

[Strauß 1993a] Friedrich Strauß: *Situation Oriented Description of User Interfaces*, in: Patrick Brezillon (Hrsg.): *Proceedings of the IJCAI'93 (13^{th} International Joint Conference on Artificial Intelligence) Workshop on „Using knowledge in its Context"*, Rapport Interne du LAFORIA (Laboratoire Formes et Intelligence Artificielle) 93/13, Institut Blaise Pascal, Paris, 1993

[Strauß 1993b] Friedrich Strauß: *Contextsensitive Help–facilities in GUIs through Situations*, in: T. Grechenig, M. Tscheligi (Hrsg.): *Proceedings of the VCHCI'93 (Vienna Conference on Human Computer Interaction)*, Lecture Notes in Computer Science 733, Springer–Verlag, Wien, 1993

[Suchman 1987] Lucy A. Suchman: *Plans and Situated Actions: The Problem of Human–Machine Communication*, Cambridge University Press, 1987

[Sukaviriya et al. 1993] Piyawadee „Noi" Sukaviriya, Martin Frank, Anton Spaans, Todd Griffith, Krishna Bharat, Jeyakumar Muthukumarasamy: *A Model–Based User Interface Architecture: Enhancing a Runtime Environment with Declarative Knowledge*, Technical Report GIT–GVU–93–12, Georgia Institute of Technology, Atlanta, Georgia, 1993

[Waite, Goos 1984] William W. Waite, Gerhard Goos: *Compiler Construction*, Texts and Monographs in Computer Science, Springer–Verlag, 1984

[Winter 1994] Kirsten Winter: *Kontextsensitive Benutzerunterstützung in graphischen Benutzungsoberflächen*, Diplomarbeit, Universität Erlangen–Nürnberg und Institut für Informatik und Gesellschaft, Universität Freiburg, 1994 (in Vorbereitung)

[Yahya, Henschen 1985] Adnan Yahya, Lawrence J. Henschen: *Deduction in Non–Horn Databases*, in: *Journal of Automated Reasoning* No. 1, 1985