T U M

I N S T I T U T F Ü R I N F O R M A T I K

Frisco STDA – Werkzeug zur methodischen Bearbeitung von Automaten

Michael Fahrmair, Bernhard Rumpe

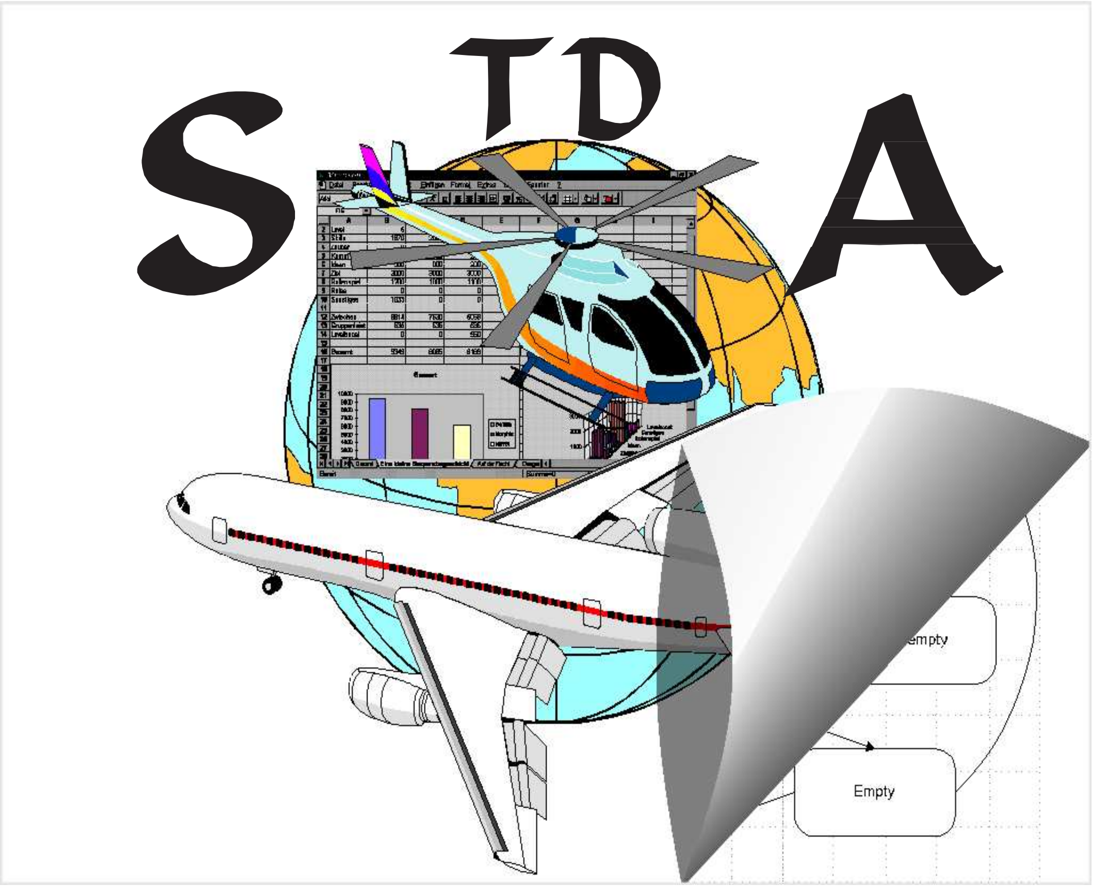
TUM-I9815
Juni 98

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-06-I9815-2/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Technische Universität München
Institut für Informatik
Lehrstuhl IV - Software Engineering
Prof. Dr. Manfred Broy



Frisco STDA
Werkzeug zur methodischen Bearbeitung von Automaten

Michael Fahrmair, Bernhard Rumpe

In diesem Bericht ist die Entwicklung eines Werkzeuges zum Einsatz von Automaten in der Softwareentwicklung beschrieben.
Basierend auf einer existierenden formalen Methodik des Entwurfs verteilter objektorientierter Systeme[33] wird ein Teilaspekt dieser Methodik, die Beschreibung von Klassen mittels Verhaltensautomaten, implementiert.
Das Werkzeug wurde als Komponente für die Werkzeugplattform OEF[30] realisiert und greift, soweit möglich, auf bereits bestehende Werkzeugkomponenten zurück, die in einer neu entwickelten Komponentenarchitektur, welche die Austauschbarkeit von Einzelkomponenten ermöglicht, vereint sind.

Diese Arbeit umfaßt:

- Eine informelle Einführung in die der Arbeit zugrundeliegende formale Methodik (Theorie).
- Tutorial zum Erlernen der komplexen Werkzeugfunktionen.
- Ausführliche Bedienungsanleitung des Werkzeugs.
- Die Beschreibung einer allgemein für komplexe OEF-Werkzeuge anwendbaren Architektur, welche einen Model-View-Mechanismus in einer Umgebung hochgradig wiederverwendbarer Komponenten realisiert und die praktische Implementierung für das konkrete Werkzeug.
- Theoretische Betrachtungen über das Parsen von Diagrammen, d.h. gemischt graphischer und textueller Information.
- Ableitung und Implementierung eines einfachen Lösungsansatzes zum Parsen graphischer Informationen für den konkreten Fall der Automatendiagramme.
- Anpassung eines existierenden Parsers (FriscoF) zum exemplarischen Parsen des Textanteils von Automatendiagrammen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einführung

1.1. Einleitung

Es mag wie ein ironischer Anachronismus anmuten, in einer Zivilisation zu leben, welche einerseits die Schwelle vom Atom- zum Informationszeitalter bereits eindeutig überschritten hat, andererseits in der Schlüsseltechnologie dieses Zeitalters, nämlich der Informationsverarbeitung, noch immer mit teilweise heuristischen Methoden und simplen Werkzeugen arbeitet, die zum guten Teil aus den frühen Anfängen dieser Periode stammen und nur wenig bis unzureichend von der zur Informationstechnologie zugeordneten Ingenieursdisziplin, der Softwaretechnik, durch wissenschaftlich fundierte Konstruktions- und Verfahrensweisen unterstützt wird.

Gleichzeitig nimmt der Bedarf an informationstechnischen Anwendungen und auch deren Umfang stetig und in fast atemberaubenden Tempo zu. Hier sei nur das Gebiet der Telekommunikation mit Millionen von Einzelabrechnungen pro Monat und hochkomplexen Vermittlungssteuerungen, besonders im Bereich der mobilen Kommunikation, Telemetrie und Steuerungsanwendungen in der Luft- /Raumfahrt und nicht zuletzt die Simulationstechnik genannt, die über Zuverlässigkeit von Kernwaffen, Atomreaktoren und Automobilen entscheidet. Hinzu kommen jede Menge neuer Anwendungen, wie elektronischer Handel, oder eine automatische Steuerung des Individualverkehrs, um nur zwei Schlagworte zu nennen.

Für alle diese Beispiele werden teilweise monolithische Programme benötigt, die nach wie vor im Grunde genauso erstellt werden, wie in der Steinzeit der Informationstechnik. Zwar hat sich partiell einiges verbessert, vor allem auf den Gebieten der Projektplanung, -Organisation und Logistik, entscheidende Vorteile bei der Berücksichtigung des 'Menschlichen Faktors'[1], wie die grundsätzliche Veranlagung einer neuronalen intuitiv-kreativen Informationsverarbeitungseinheit, genannt menschliches Gehirn, zum Begehen von Fehlern euphemistisch umschrieben wird, wurden jedoch nicht erzielt.

An dieser Stelle kommen dann effektive, aber wie die Erfahrung lehrt sehr teure Verfahren zum Einsatz, die oft unter dem Begriff Qualitätsmanagement zusammengefaßt werden. Dahinter verbirgt sich dann nichts weiter, als der verzweifelte

[1] engl. *human factor*

Versuch, alle plausibel erscheinenden Fälle zu testen. Durch Einführung weiterer Redundanz oder komplizierter Werkzeuge wird dann versucht, Fehler beim Testen zu vermeiden, natürlich durch den Einsatz von Menschen und Werkzeugen, deren Fehlerfreiheit keineswegs gesichert ist.
Auf der anderen Seite des Problems stehen dann unter anderem eingangs aufgeführte Beispiele von Anwendungen, in denen allesamt bereits der kleinste Fehler katastrophale Auswirkungen haben kann. Eine falsche Telefonrechnung ist noch ein harmloser Fall, ein Bordcomputer eines Airbus, der für den Fall eines doppelten Triebwerksbrandes ein Verfahren empfiehlt, das zur Deaktivierung der Sauerstoffversorgung der Passagierkabine führt, oder der Überlauf eines Meßwertes im Telemetriesystem der Ariane 5, der ein ehrgeiziges Projekt in ein mehrere hundert Millionen Feuerwerk verwandelt, wiegt da weit schlimmer. Auch die potentielle Möglichkeit einer dank eines Softwarefehlers falsch gelaufenen Finanztransaktion in Millionenhöhe rechtfertigt die Entwicklung neuer Lösungsansätze.
Um diese Lösungsansätze zu finden, ist es oft hilfreich, Anleihen bei etablierten Ingenieurwissenschaften zu machen, deren korrespondierende Techniken bereits seit mehreren hundert Jahren Anwendung finden. Das Wort Lösungsansatz ist dabei wohl mehr ein Ausdruck des Wunsches, als ein Spiegelbild der Realität. Jedem dürfte klar sein, daß es kein Verfahren gibt, Fehler bei der Erstellung und beim Ablauf eines beliebig komplexen Systems auszuschließen, solange keine hundertprozentig kontrollierte Ablaufumgebung existiert. Es geht vielmehr um die Vermeidung von Fehlern und damit eigentlich mehr um eine Linderung des Problems, als um eine Lösung.
Ein enger Verwandter zur Wissenschaft der Softwareentwicklung ist die Architektur mit der Bautechnik als korrespondierender Anwendung. Hier werden anstelle von Programmen Bauwerke errichtet, deren Umfang und Komplexität fast beliebig sind. Auch hier treten mit zunehmendem Umfang immer größere Schwierigkeiten auf und auch hier kann bereits ein kleiner Fehler fatale Auswirkungen, nämlich den Einsturz des Gebäudes, zur Folge haben.
Auch die Bautechnik war lange Zeit in der Lage, beeindruckende Bauwerke zu erstellen, ohne über mehr als rudimentäre theoretische Grundlagen und aus heutiger Sicht völlig unzureichende Werkzeuge und Verfahren zu verfügen. Auch hier wurden alle Unzulänglichkeiten durch einen immensen Aufwand an Ressourcen und großartige Fähigkeiten auf dem Gebiet der Logistik und Organisation teilweise wettgemacht, die im Laufe der Zeit ständig verbessert wurden.
Verzichteten die alten Ägypter noch auf Komplexität zugunsten der Errichtung von Bauwerken gewaltigen Umfangs, wurden in Europa zunächst viele der komplexeren Bauwerke nach dem Prinzip von Versuch und Irrtum errichtet. Stürzte ein Gebäude aufgrund mangelhafter Statik ein, wurde es einfach wieder neu (und anders) aufgebaut.
Erst in der heutigen Zeit existieren theoretisch fundierte Konstruktions- und Prüfverfahren, die, unterstützt durch eine Bandbreite von Werkzeugen, in jedem Schritt des Bauprozesses von der Idee, über die Planung, Ausführung und

Qualitätssicherung durchgängig Anwendung finden.
Ähnliches gilt auch für alle anderen Ingenieurwissenschaften wie Elektrotechnik und Maschinenbau. Die Softwareentwicklung leidet aufgrund der Tatsache, daß es sich um eine sehr junge Disziplin handelt, die zu einer ebenfalls noch sehr jungen Technik korrespondiert, natürlich an vielen Stellen noch an nicht akzeptablen Unzulänglichkeiten. Es besteht allerdings die Möglichkeit, wegen der Verwandtschaft zu anderen Ingenieurwissenschaften, dort durch Analyse Verfahren abzuleiten, und sie dann an die eigenen Bedürfnisse anzupassen.
Durch diesen Umstand kann der sehr langwierige Prozeß der Etablierung stark beschleunigt werden, schließlich will kein Mensch 4000 Jahre warten, bis eine Flugreise endlich in einem Flugzeug stattfinden kann, dessen Leitsysteme nach allen Regeln der Kunst sicher sind.

1.2. Motivation

Beim Vergleich der Softwareentwicklung mit klassischen Ingenieurswissenschaften und bei Analyse der Unterschiede unter dem Gesichtspunkt der Fehlervermeidung, kann eine interessante Beobachtung gemacht werden: es wird versucht, den menschlichen Faktor in jedem Abschnitt der angewandten (und fest vorgegebenen) Verfahren zu berücksichtigen. Bereits bei der Planung kommen feste Konstruktionsregeln zum Einsatz, deren Korrektheit meist (über die Physik) bewiesen wurde.
Bei der Herstellung werden diese Ergebnisse dann möglichst ohne unbeaufsichtigte menschliche Einflußnahme in ein fertiges Produkt umgesetzt. Wo dies nicht, oder nur unzureichend möglich ist, greifen sofort im Anschluß genormte Prüfverfahren. Das fertige Produkt wird dann wiederum einer Gesamtprüfung unterzogen.
In der Softwareentwicklung existieren zwar ebenfalls genormte Verfahrensweisen und Ansätze zu einer automatisierten Endkontrolle, dazwischen fehlt es aber an vielen Ecken und Enden. Bereits während des Planungsprozesses fehlt es an Werkzeugen zur Fehlerprüfung und das, obwohl gerade Fehler in einer frühen Phase des Projektes zu den höchsten Kosten führen, was eine der wichtigsten Erkenntnisse in der Softwareentwicklung darstellt.
Auch die Übernahme der Ergebnisse der Planung in den der Produktion in anderen Ingenieursdisziplinen verwandten Prozeß der Programmerstellung erfolgt in der Regel ohne Aufsicht durch eine in ein Werkzeug implementierte Verfahrenstechnik.
Der Programmierer erhält die Vorgaben aus dem Planungsprozeß, d.h. die Spezifikation und setzt diese dann eigenständig in Programmcode um. An dieser Stelle ist bereits keine durchgehende Verfahrensprüfung mehr gegeben. Ein Mensch sorgt für die Umsetzung der Spezifikation in die nächste Stufe des Verfahrens. Und wiederum Menschen sind dafür verantwortlich, zu prüfen, ob die erfolg-

te Umsetzung auch tatsächlich den Spezifikationen entspricht. In den weiteren Verfahrensschritten kann dann wieder eine automatisierte nichtmenschliche Überprüfung greifen, zumindest existieren dafür bereits Ansätze.
Für Compiler gibt es schon länger Syntax- und Kontextprüfungen, neue Sprachen wie Java legen ein sehr robustes Laufzeitverhalten an den Tag und es sind Werkzeuge in der Entwicklung, die eine noch weitergehende Verifikation von Quelltexten ermöglichen. Eine echte Lücke besteht also hauptsächlich in der Planungsphase.
Zwar existiert bereits eine fast inflationäre Anzahl an Modellen und Beschreibungstechniken aus dem Bereich der theoretischen Informatik, denen allesamt gemeinsam ist, daß sie zwar formal gut fundiert sind, sich jedoch durch eine schlechte Skalierbarkeit auf große Systeme auszeichnen, die in der Praxis der Softwareentwicklung häufig auftreten.
Aus diesen genannten Gründen erreichen diese Methoden und Modelle auch selten ein Stadium, das über die reine Theorie hinaus geht, beispielsweise in Form eines realen Werkzeuges und falls dem doch so sein sollte, scheitert es in der Regel an der Akzeptanz im industriellen Einsatz, da meist eine radikale Umstellung der Arbeitsweise gefordert wird und sie sich schlecht in bestehende Verfahren integrieren lassen.
Gleichzeitig sind in der praxisorientierten Softwaretechnik Methoden entstanden, die in der systematischen Softwareerstellung tatsächlich Verwendung gefunden haben. Es handelt sich dabei meist nur um reine Beschreibungstechniken, denen aber kein mathematisch fundiertes Modell zugrunde liegt. Allzuoft ist auch die Semantik der Beschreibungstechniken nicht exakt definiert und eine Integration der durch verschiedene angewandte Techniken definierten Sichten und die damit verbundenen Konsistenzbedingungen nicht ausreichend geklärt.
Unter dem Gesichtspunkt der Fehlervermeidung schneiden diese Verfahren noch schlechter ab. Aufgrund der mangelhaften formalen Fundierung und nicht exakten Semantik existieren nur begrenzte Möglichkeiten, Fehler in der Spezifikation automatisch durch ein Werkzeug aufzuspüren. Die Erstellung einer Spezifikation wird dagegen noch lange Zeit ein kreativer Prozeß bleiben, da mit einer automatisierten Umsetzung der Anforderungen in nächster Zeit sicher nicht zu rechnen ist.
Ein kreativer Prozeß bedeutet aber unweigerlich, daß dabei Fehler entstehen können, da menschliche Denkprozesse dazu neigen, Dinge zu übersehen, oder nicht geklärte Sachverhalte als gegeben zu nehmen.
Diese Fehler werden zwar meist erkannt, jedoch allzuoft erst sehr spät im Produktionsprozeß. Zu diesem Zeitpunkt sind die Fehler dann nur sehr schwer zu beheben und 'sehr schwer' ist in diesem Falle gleichbedeutend mit 'sehr teuer'.
Auch können Fehler bei der Umsetzung der Spezifikation entstehen, da diese Umsetzung nicht automatisiert erfolgt, sondern durch einen Menschen. Gerade an dieser Stelle kann es dann aufgrund der Schwächen in der Semantik zu Fehlinterpretationen kommen.

Wünschenswert ist ein Werkzeug, das, basierend auf einem formal fundierten Modell, ein oder mehrere Beschreibungstechniken zur Verfügung stellt. Für diese muß eine Möglichkeit bestehen, automatisiert bestimmte grundlegende Eigenschaften, wie Richtigkeit (sowohl formale Richtigkeit, als auch semantische Konsistenz), Vollständigkeit und Erfüllbarkeit zu prüfen und das Ergebnis dann ohne weitere menschliche Intervention in Quelltext zu übersetzen.
Ziel der vorliegenden Arbeit ist es jetzt, basierend auf einer existierenden theoretischen Arbeit über eine 'Formale Methodik des Entwurfs verteilter objektorientierter Systeme' ([33]) Architektur und Kernstück eines Werkzeuges zu entwickeln, das eine der in besagter Arbeit beschriebenen Beschreibungstechniken zum Entwurf von Komponenten implementiert und zwar so, daß es den oben gestellten Anforderungen wesentlich besser gerecht wird, als andere existierende Werkzeuge.
Die verwendete formale Methodik hat den Vorteil, daß sie sich stark an praktischen Techniken orientiert. Sie kombiniert in der Praxis bereits eingesetzte graphische Beschreibungstechniken mit formalen Ansätzen und vereint so die Vorteile beider Techniken.
Die Beschreibungstechniken erhalten eine präzise formale Semantik und es werden syntaxbasierte Entwicklungsschritte formalisiert, die dann als relativ zur Semantik korrekt bewiesen werden. Somit wird eine eigenschaftserhaltende Konstruktion auf Basis einer Beschreibungstechnik möglich. Ein weiterer nützlicher Effekt, der eine Konstruktion begleitet, ist die Tatsache, daß plötzlich ein Fortschritt meßbar ist, ein Umstand der die Projektplanung und logistische Organisation erleichtert.
Das entwickelte Werkzeug ist Teil der bereits bestehenden Softwareentwicklungsumgebung OEF[2]. Das OEF ist eine einheitliche Laufzeitumgebung für Werkzeuge aus dem Bereich der Softwareentwicklung, darüber hinaus auch ein Framework für die Entwicklung von Softwareentwicklungswerkzeugen. Im Gegensatz zu anderen Plattformen wie z.B. AutoFocus, verwendet das OEF einen dokumentenorientierten Ansatz, d.h. jede weitere Informationsverarbeitung findet grundsätzlich auf der Basis der als Dokumentation zu einem Projekt vorliegenden Informationen (Tabellen, Formeln, Diagramme) statt.
Vorteile dieses Ansatzes sind unter anderem, daß die zur Eingabe notwendige Arbeitsweise so nah wie möglich an traditionell üblichen Verfahrenstechniken im Verlauf der Dokumentationserstellung (Papier & Bleistift, bzw. Textverarbeitung, Tabellenkalkulation und Grafikprogramm) liegt, ohne daß auf die Vorteile verzichtet werden müßte, die spezialisierte Software bieten kann (z.B. Kontext- und Regelüberprüfungsmechanismen). Zusätzlich besteht die Möglichkeit der Automatisierung des Konstruktionsprozesses, so daß nicht nur Unterstützung bei der sauberen Dokumentation von Ergebnissen erfahren wird, sondern auch bei der Durchführung des Prozesses an sich, welcher zu den gewünschten Ergebnissen führt und eine Hilfestellung für die saubere Dokumentation der Zwischenschritte

[2]Open Editor Framework

bereitgestellt wird. Weitere Informationen können anderen Dokumentationen zu diesem Thema entnommen werden [30, 6].
Um eine prinzipiell unbeschränkte Auswahl an Beschreibungstechniken zu ermöglichen, besteht ein OEF-Dokument aus einzelnen Parts.
Ein *Part* ist die sichtbare Darstellung einer eigenständigen Werkzeugkomponente, genannt *Parthandler*. Einfache Parthandler sind auf die Darstellung und ggf. das Editieren einer bestimmten Informationsart spezialisiert. Dabei kann es sich um einfachen Text genauso wie um Grafiken, Tabellen und Animationen handeln.
Komplexere Parthandler (im folgenden *Documentcontroller* genannt) verwalten und verknüpfen mehrere Werkzeugkomponenten innerhalb eines Dokumentes zu einer komplexen Werkzeuganwendung und führen damit das sie umgebende Dokument einem speziellen Verwendungszweck zu, der über die reine Dokumentation hinausgeht.

1.3. Ein Beispiel

Zur Verdeutlichung der motivierten Thematik und um eine Vorstellung zu vermitteln, was in den folgenden Kapiteln dieser Arbeit genauer beschrieben wird, soll an dieser Stelle ein erstes kleines Beispiel dienen, das unter OEF mit dem Werkzeug STDA[3](dem Kernstück dieser Diplomarbeit) erstellt wurde.

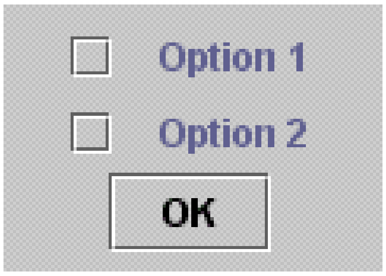


Abbildung 1.1.: Beispieldialog

Es handelt sich um die Teilkonstruktion eines Dialogs (Abb. 1.1). Aus Gründen der Übersichtlichkeit wird angenommen, daß alle notwendigen Initialisierungen der einzelnen Bedienungselemente bereits erfolgt sind. Die Beispielskomponente beschreibt also nur das Verhalten des Dialogs und für das Beispiel wird nur ein Teilaspekt davon modelliert. Der Dialog gibt zwei Optionen zur Auswahl, wobei es nicht möglich ist, die zweite Option zu deaktivieren, sobald sie einmal aktiviert wurde.
Um das Verhalten des Dialogs zu beschreiben, wird in diesem einfachen Fall am besten mit vier Zuständen begonnen, die alle Möglichkeiten des Datenzustandsraums repräsentieren (Abb. 1.2).
Weiterhin ist zunächst nicht sicher, ob die Aktivierung der zweiten Option Sinn macht, wenn nicht auch gleichzeitig die erste Option aktiviert wurde. Im Sinne einer robusten Modellierung soll als Alternative zum getrennten Aktivieren auch einen Möglichkeit modelliert werden, welche die erste Option einfach mitaktiviert. Dazu wird zunächst eine weitere Transition hinzugefügt, die sofort in den Zustand `1&2` überführt, wenn die zweite Option aktiviert wird.

[3]State-Transition-Diagram-Assistent

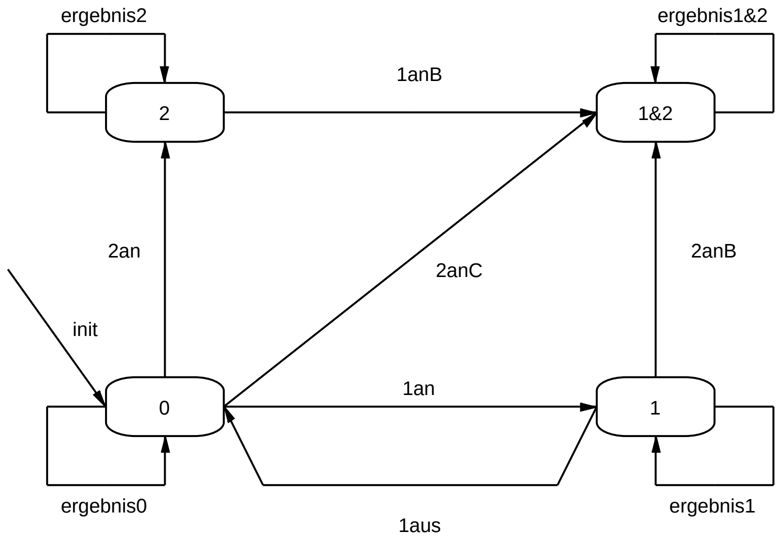


Abbildung 1.2.: Beispielautomat (Grundmodell)

State	State-Pattern	State-Predicate
2		
1&2		
1		
0		

Tabelle 1.1.: Zustandstabelle des Beispiels

Transition	Input	PreC	Output	PostC	Initial
erg.1&2	Request(s)		[Respond(s,true,true)]		false
ergebnis2	Request(s)		[Respond(s,false,true)]		false
ergebnis0	Request(s)		[Respond(s,false,false)]		false
ergebnis1	Request(s)		[Respond(s,true,false)]		false
1anB	An(1)				false
2anB	An(2)				false
2anC	An(2)				false
2an	An(2)				false
1an	An(1)				false
1aus	Aus(1)				false
init					true

Tabelle 1.2.: Transitionstabelle des Beispiels

Nachdem das Grundmodell erstellt wurde, wird es nach festen Regeln verfeinert, zuvor werden jedoch bestimmte Bedingungen überprüft, beispielsweise ob alle Zustände erfüllbar und alle Transitionen schaltbereit sind. Dazu sind die automatisch generierten Beweisverpflichtungen aus Tabelle 1.3 zu verifizieren. Die festen Verfeinerungsregeln sorgen dafür, daß sich an diesen Bedingungen nichts ändert, bzw. daß sie für die geänderten Stellen neu bewiesen werden.
Das Verhalten des Grundmodells soll nun verfeinert werden. Es wurde nachträglich entschieden, daß eine Aktivierung der zweiten Option ohne die erste Option keinen Sinn macht. In Zustand `0` stehen für die Behandlung einer identischen Eingabenachricht innerhalb eines Zustandes zwei Möglichkeiten zur Verfügung. Mit einem Verfeinerungsschritt kann nun eine der beiden Transitionen (nach der Entscheidung also `2an`) entfernt werden, wobei natürlich bewiesen werden muß, daß eine andere Transition existiert, welche die Aufgabe der entfernten Transition übernimmt (siehe auch Abb. 1.3 und Tabelle 1.4).

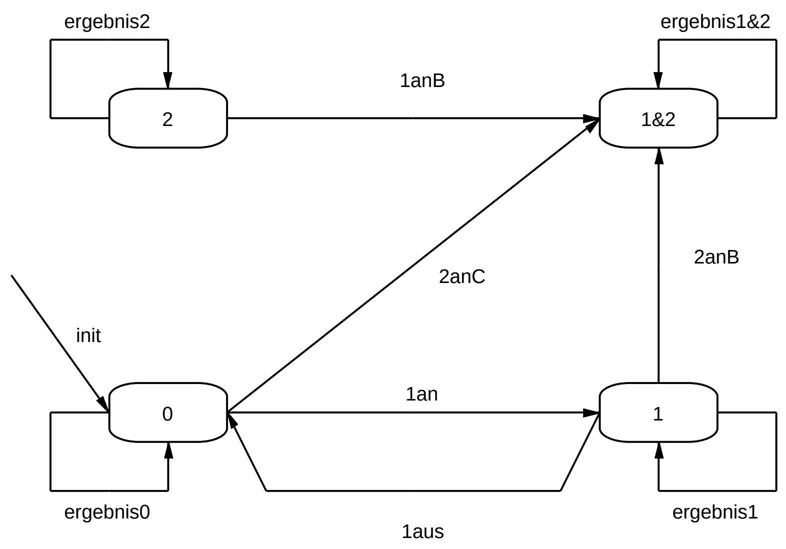


Abbildung 1.3.: Beispielautomat (1. Verfeinerungsschritt)

Durch das Löschen der Transition wird aber der Zustand `2` unerreichbar und kann deshalb ebenfalls mitsamt all seiner Transitionen entfernt werden. Auch hier ist der notwendige Beweis sehr einfach, er ist sozusagen im Diagramm ‘sichtbar’ (Abb. 1.4).
Insgesamt stehen acht Verfeinerungsschritte zur Verfügung, jeweils das Hinzufügen, Löschen und Verfeinern (Detaillieren) von Zuständen und Transitionen sowie die Entfernung und Verfeinerung von Initialelementen.

```
ax {
  2. TT;
  1&2. TT;
  1. TT;
  0. TT;
}

ax ALL inp::IN .{
  ergebnis1&2. (EX (s) :: Object.inp = Request(s)) =>
               (EX (s) :: Object,out::[OUT].
                inp = Request(s) AND out=[Respond(s;true;true)]);
  ergebnis2. (EX (s) :: Object.inp = Request(s)) =>
             (EX (s) :: Object,out::[OUT].
              inp = Request(s) AND out=[Respond(s;false;true)]);
  ergebnis0. (EX (s) :: Object.inp = Request(s)) =>
             (EX (s) :: Object, out::[OUT].
              inp = Request(s) AND out=[Respond(s;false;false)]);
  ergebnis1. (EX (s) :: Object.inp = Request(s)) =>
             (EX (s) :: Object, out::[OUT].
              inp = Request(s) AND out=[Respond(s;true;false)]);
  1anB. (EX (1) :: Int.inp = An(1)) =>
        (EX (1) :: Int, out::[OUT].
         inp = An(1) AND out=[]);
  2anB. (EX (2) :: Int.inp = An(2)) =>
        (EX (2) :: Int, out::[OUT].
         inp = An(2) AND out=[]);
  2anC. (EX (2) :: Int.inp = An(2)) =>
        (EX (2) :: Int, out::[OUT].
         inp = An(2) AND out=[]);
  2an. (EX (2) :: Int.inp = An(2)) =>
       (EX (2) :: Int, out::[OUT].
        inp = An(2) ND out=[]);
  1an. (EX (1) :: Int.inp = An(1)) =>
       (EX (1) :: Int, out::[OUT].
        inp = An(1) AND out=[]);
  1aus. (EX (1) :: Int.inp = Aus(1)) =>
        (EX (1) :: Int, out::[OUT].
         inp = Aus(1) AND out=[]);
};
```

Tabelle 1.3.: Beispiel: Automatisch generierte Beweisverpflichtungen

```
ax ALL inp::IN .{
  2an. (EX (2) :: Int . inp = An(2)) =>
       ((EX (s) :: Object . inp = Request(s)) OR
       (EX (2) :: Int . inp = An(2)) OR
       (EX (1) :: Int . inp = An(1)));
};
```

Tabelle 1.4.: Beispiel: Beweisverpflichtungen (Verfeinerungsschritt)

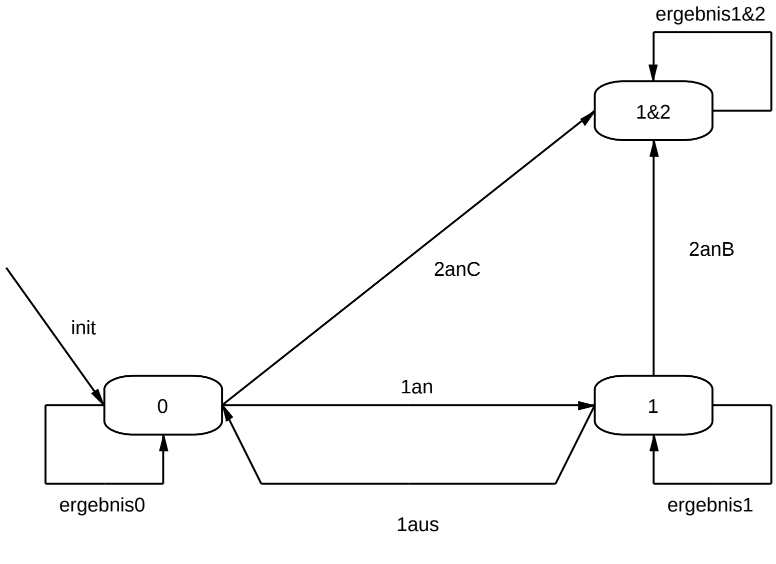


Abbildung 1.4.: Beispielautomat (2. Verfeinerungsschritt)

1.4. Struktur dieser Arbeit

Nach Einführung und Motivation in diesem Kapitel folgt ab Seite 13 ein Kapitel, welches das notwendige theoretische Grundwissen vermittelt. Es handelt sich dabei im wesentlichen um ein rein informelles Exzerpt von [33], das eine formale Methodik zur Entwicklung verteilter objektorientierter Systeme enthält. Wer mit dieser Methodik bereits vertraut ist, kann das meiste überspringen und direkt mit Abschnitt 2.5 beginnen.
Das dritte Kapitel ab Seite 35 ist ein direkter Einstieg in die Bedienung des Werkzeuges STDA zur Bearbeitung von Automatendokumenten. Es enthält eine schrittweise Einführung in die Werkzeugbedienung und ein kleines Tutorial.
Das Kapitel 4 ab Seite 65 enthält mit der Aufzählung der Anforderungen an das zu entwickelnde Werkzeug die Beschreibung der eigentlichen Aufgabenstellung dieser Arbeit samt notwendiger Erläuterungen.
Das fünfte Kapitel ab Seite 79 ist in zwei Themenblöcke unterteilt. Der erste Teil richtet sich auch an technisch interessierte Anwender. Er enthält eine Betrachtung des Gesamtsystems und der notwendigen Entwurfsentscheidungen. Im Anschluß daran werden Lösungen für interessante bzw. neuartige Probleme entwickelt, die in Zusammenhang mit der Entwicklung des Werkzeuges STDA auftraten. Der zweite Teil ab Seite 98 richtet sich dann ausschließlich an Programmierer, die Wartungsarbeiten, Änderungen oder Erweiterungen vornehmen wollen und dafür eine technische Dokumentation benötigen. Er enthält eine ausführliche Beschreibung und Implementierungsdetails zu den einzelnen Komponenten des Gesamtsystems.
Die Arbeit schließt mit Kapitel 6 ab Seite 123 und einer kritischen Bewertung der realen Anwendbarkeit des Werkzeugs sowie der darin implementierten Entwicklungsmethodik und einer kurzen Analyse über die Chancen und Möglichkeiten zur Etablierung auf dem industriellen, d.h. nichtuniversitären Markt.

2. Theoretische Grundlagen

Dieses Kapitel enthält in informeller Weise alle theoretischen Grundlagen, die über informationstechnisches Grundwissen hinausgehen, zum Verständnis der folgenden Kapitel aber unbedingt notwendig sind.
Es handelt sich dabei im wesentlichen um ein Exzerpt von [33], das eine formale Methodik zur Entwicklung verteilter objektorientierter Systeme beschreibt. Es sind aber nur die für diese Arbeit wichtigen Abschnitte über Automatendokumente und deren formale Grundlagen behandelt. Wer mit der Materie bereits vertraut ist, kann das meiste überspringen und direkt mit Abschnitt 2.5 beginnen.
Wie bereits erwähnt, ist bei der Aufbereitung der Informationen, vor allem zu diesem Kapitel auf leichte und schnelle Verständlichkeit geachtet worden. Wer Wert auf eine jenseits des Umfangs dieser Arbeit liegende tiefergehende Betrachtung der einzelnen Themen legt, sei auf die angegebene Literatur verwiesen.

2.1. Eine formale Methodik

Grundlage dieser Arbeit ist eine formale Methodik für den Entwurf verteilter objektorientierter Systeme. Die wichtigsten Begriffe dieser Methodik werden im folgenden kurz erklärt.
Zentraler Begriff dieser Methodik ist das Dokument. Ein *Dokument* dient zur Beschreibung gewisser Aspekte oder Ausschnitte eines zu erstellenden Software-/Hardwaresystems. *Ausschnitt* eines Systems ist dabei ein Systemteil, z.B. eine Komponente, oder ein Subsystem. Ein *Aspekt* charakterisiert eine bestimmte Systemsicht, z.B. die Kommunikationsstruktur innerhalb der Komponenten.
Während der Systementwicklung werden unterschiedliche Arten von Dokumenten erstellt und verschiedenartig genutzt. Dabei sorgt eine *Syntax* für eine eingeschränkte Konsistenz und eine *Semantik* beschreibt die Bedeutung des Dokumentes.
Die Methodik definiert verschiedene Dokumentarten sowie Beweise für die Korrektheit der zugehörigen Entwicklungsschritte. Die definierten Dokumentarten sind vor allem für die frühe Entwurfsphase einer Systementwicklung prädestiniert.

- *Datentypdefinitionen* sind ein wichtiges Beschreibungsmittel für den Systementwurf. Grundlegende im System genutzte Datentypen wie Zahlen, Namen oder Verbunde werden nicht axiomatisch charakterisiert, sondern als funktionale Datentypen formuliert.

- *Klassenbeschreibungen* ist eine Dokumentart, die in den Komponenten einer Klasse gekapselte Datenstruktur in Form einer Menge von Attributen und deren Sorten sowie die Methodensignatur und die Anzahl der erlaubten Instanzierungen beschreibt.

- *Objektmodell* ist eine Dokumentart, die Beziehungen der Daten des Systems, deren Beziehungen untereinander, durch entsprechende Attribute realisierte Datenabhängigkeiten und Vererbungen durch eine graphische Repräsentation beschreibt.

- *Lebenszyklen (Automaten)* ist eine Art von Dokumenten, die mögliche Zustandsübergänge von Elementen einer Klasse bei der Verarbeitung von Nachrichten und die Reaktionen darauf beschreibt. Ein Lebenszyklus kann sehr detailliert sein, womit er sich dann direkt zur Ausführung als Prototyp eignet, oder aber sehr abstrakt, wenn nur Äquivalenzklassen der Eingabe-, Ausgabenachrichten und der Zustandsmenge betrachtet werden.

Die letzte Dokumentart ist Basis für das in dieser Arbeit entwickelte Entwurfswerkzeug. Je nach Art der verwendeten Beschreibungstechniken können während der Systementwicklung komplexe Beziehungen zwischen einzelnen Dokumenten entstehen. Die Methodik enthält eine formale Semantik, bezeichnet als *Systemmodell*, welche die einzelnen Semantiken der verschiedenen Dokumentarten integriert, damit eine präzise Formulierung von Kontextbedingungen zwischen den Dokumenten und eine semantikerhaltende Übersetzung eines Dokumentes (*Generierung*) möglich wird. Die Methodik beschäftigt sich weiterhin mit der zielgerichteten Anwendung von Entwicklungsschritten in der Systementwicklung.
Ein *System* ist eine konkrete Ausprägung aus Hard- und Softwarekomponenten sowie gegebenenfalls menschlichen Akteuren, mit einer bestimmten Struktur und bestimmten Verhalten. Ein *Systemablauf* ist eine Abfolge von Eingaben, Ausgaben und Zuständen für jeden relevanten Zeitpunkt der Abfolge. Ein nichtdeterministisches System kann zu gleichen Eingaben und Anfangszuständen verschiedene Abläufe besitzen. Das *Systemmodell* ist die Charakterisierung aller interessanten Systeme in Verhalten und Struktur, jedoch mit Freiheitsgraden, die durch eine konkrete Systementwicklung spezialisiert werden. Das Systemmodell wird in zwei Schichten definiert. Ein *Kern* definiert die Abläufe von Systemen und führt den Begriff des Agenten ein. Die darauf aufgesetzte zweite Schicht führt durch Einführung von Klassen eine Verbindung zwischen der semantikorientierten Welt des Systemmodells und der syntaktisch orientierten Modellierung (Beschreibung) eines Systems ein. Das Systemmodell dient also zur semantischen Fundierung von Techniken zur Beschreibung verteilter objektorientierter Software.

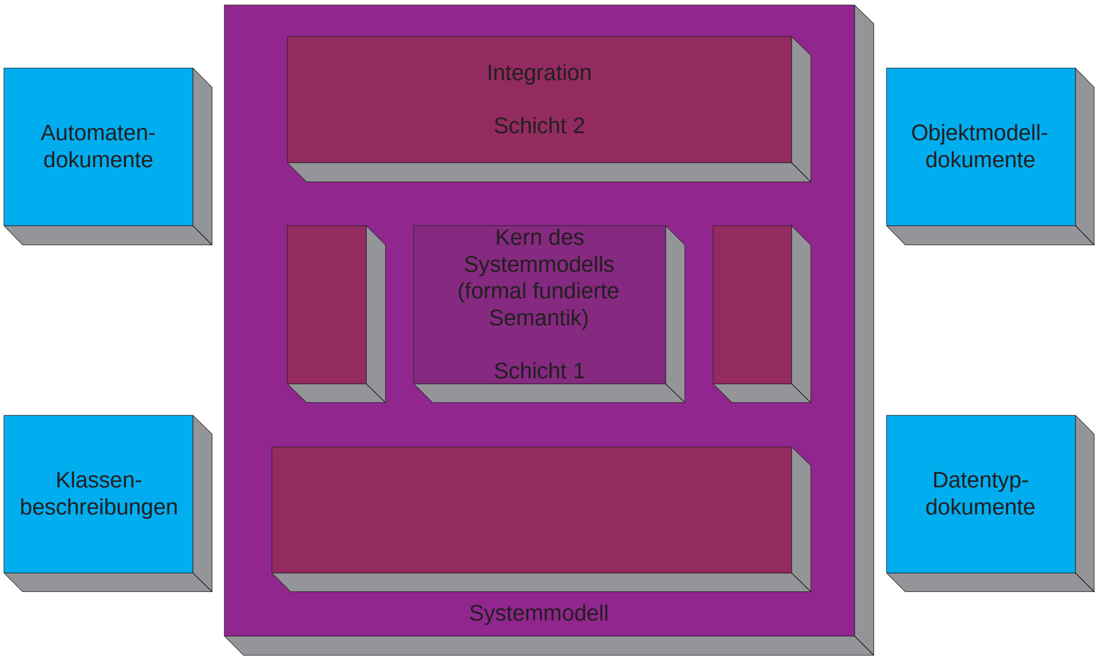


Abbildung 2.1.: Das 2-Schichten-Systemmodell

Ein objektorientiertes System besteht aus Agenten, die parallel interagieren können. Ein *Agent* ist eine konzeptuelle Einheit, die über einen gekapselten Speicher und eine festgelegte Schnittstelle verfügt. Ein gemeinsamer Speicher für Agenten ist nicht vorhanden, da sie konzeptionell oder physisch verteilt sein können. Jede Interaktion findet dagegen durch Nachrichtenaustausch statt. *Nachrichten* sind informationstragende Einheiten für die Kommunikation zwischen Agenten. Die gesamte Kommunikation ist *asynchron*, d.h. der Sender darf eine Nachricht versenden, unabhängig von der Empfangsbereitschaft des Empfängers. Das Konzept von Agenten ist eng an Objekte in objektorientierten Systemen angelehnt. Der Name Agent drückt zusätzlich eine eigenständige Aktivität der Komponenten aus, im Gegensatz zu den eher passiven Objekten. Agenten werden dynamisch erzeugt und zerstört, ebenso wie ihre Kommunikationswege, sogenannte *Kanäle*. Kommunikationswege werden mittels Identifikatoren ermöglicht, die den Empfänger einer Nachricht identifizieren. Ein Kanal besteht, wenn dem Sender der Identifikator eines Empfängers bekannt ist. Das Systemmodell läßt die dynamische Erzeugung von Agenten zu, die Zerstörung wird nicht explizit modelliert, da diese wie zum Beispiel in Java von einem Garbage Collector erledigt werden kann.

Einem Agent ist sein eigener Identifikator bekannt. Die unbeschränkte Menge der Agenten wird in Klassen gruppiert, dadurch entstehen Äquivalenzklassen und die potentiell unendliche Menge von Agenten kann mit endlich vielen Dokumenten beschrieben werden. Klassen bilden damit eine Brücke zwischen den klassenbasierten Beschreibungstechniken und dem Kern des Systemmodells, das agentenbasiert ist (Schicht 2). Für eine detailliertere Beschreibung des Begriffsmodells und eine vollständige Charakterisierung des verwendeten Systemmodells siehe [33] Kapitel 2 und 3.

2.2. Automaten

Ein Automat beschreibt das Verhalten einer Komponente, indem zu jedem Zustand der Komponente die Reaktion auf jede Eingabe angegeben wird. Die Reaktion besteht aus Verarbeitung eines Eingabezeichens, der Produktion einer Sequenz von Ausgaben und dem Übergang in einen neuen Zustand, von dem das nächste Eingabezeichen verarbeitet wird. Die bei einer Transition stattfindende Ausgabe wird daher wie bei einem Mealy-Automaten [13] der Transition zugeordnet. Der Sonderfall einer unendlichen Sequenz von Ausgaben, wird durch eine finale Transition modelliert, die mit einer unendlichen Ausgabesequenz versehen ist. Dieser Automatentyp heißt *buchstabierend*, im Gegensatz zu wortverarbeitenden Automaten [9], die einzelne Nachrichten verarbeiten. Dieser Ansatz, einen Automaten zur Beschreibung des Ein- /Ausgabeverhaltens von Komponenten zu verwenden ist verwandt mit dem Ansatz der I/O-Automaten [16, 14]. Im Unterschied dazu wird hier mit <u>einer</u> Transition sowohl die Eingabe eines Zeichens, als auch die Reaktion darauf modelliert. Dadurch werden keine Kontrollzustände benötigt, welche die Ausgabe der Zeichen kontrollieren, sondern es sind nur Datenzustände notwendig, die Äquivalenzklassen des Datenzustandsraumes der beschriebenen Komponenten darstellen.

2.2.1. Formale Fundierung

Ausgehend von buchstabierenden Automaten wird in [33] für diese zunächst eine Semantik in Form eines Prädikates über stromverarbeitenden Funktionen definiert. Für allgemeine Automaten wird analysiert, was es bedeutet, wenn in einem Zustand kein Zeichen akzeptiert werden kann, da in einem asynchronen Modell eine Einheit Nachrichten immer empfangen muß. Über die Festlegung, daß mit diesem Sachverhalt Chaos modelliert wird, gelingt es, einen allgemeinen Automaten auf einen totalen Automaten zurückzuführen, für den gezeigt werden kann, daß er konsistent ist.
Weiterhin wird in [33] gezeigt, daß es möglich ist, für jede Menge von stromverarbeitenden Funktionen einen buchstabierenden Automat mit genau dieser Menge als Semantik anzugeben.
Eine Beschreibungstechnik für Verhaltensautomaten, die auf diese Weise mit einer integrierten formalen Semantik versehen werden soll, ist also derart zu gestalten, daß sie sich in einen buchstabierenden Automaten überführen läßt (Abb. 2.2).

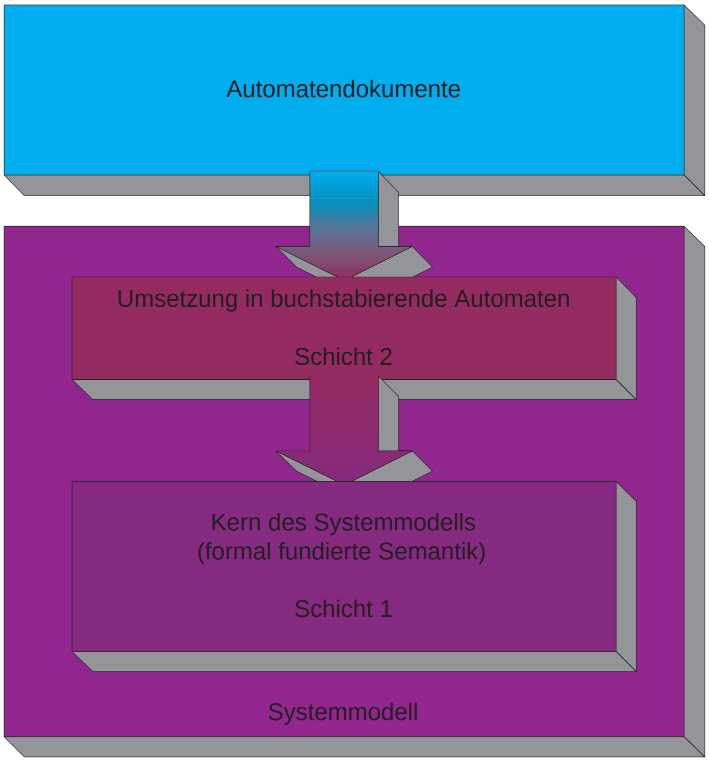


Abbildung 2.2.: Das Automatendokument im 2-Schichten-Systemmodell

2.3. Automatendokumente

Das in dieser Arbeit beschriebene Entwicklungswerkzeug implementiert die Dokumentenart ‘Automatendokument’ der in Abschnitt 2.1 beschriebenen Methodik. Dabei handelt es sich um eine Dokumentart zur Beschreibung des Lebenszyklus einer Klasse mittels Automaten. Automaten werden benutzt, um das Ein-/Ausgabeverhalten von Komponenten zu modellieren (siehe auch Abschnitt 2.2).

2.3.1. Semantik

Ein Automatendokument dient zur Charakterisierung des Verhaltens und der Zustandsfolgen eines Agenten und damit dessen Lebenszyklus.
Die Semantik eines Automatendokuments wird durch Rückführung auf die in [33] beschriebene Semantik von buchstabierenden Automaten definiert, wodurch eine zweistufige Semantikdefinition für Automatendokumente vorgenommen wird. Zunächst werden diese in einen buchstabierenden Automaten um- und dann über die formale Fundierung mit dem Systemmodell in Beziehung gesetzt.
Jeder in einem Automatendokument enthaltene Automat ist einer Klasse zugeordnet. Er beschreibt das Verhalten der zu dieser Klasse gehörenden Agenten. Dabei gibt es zwei Varianten. Ein *Klassenautomat* beschreibt das Verhalten aller Instanzen seiner Klasse, ein *Typautomat* das aller Elemente seiner Klasse.
Ein Typautomat ist allgemeiner, er beschreibt auch das Verhalten der Instanzen aus Subklassen. Ein Klassenautomat wird dagegen nicht vererbt und kann spezifischer das Verhalten der Instanzen einer Klasse beschreiben. Ein Klassenautomat wird daher als abstrakte Beschreibung der Implementierung einer Klasse betrachtet, während ein Typautomat die Schnittstelle einer Klasse und ihrer Subklassen beschreibt.

2.3.2. Konkrete Syntax

Ein Problem bei der Darstellung eines Automaten ist die Tatsache, daß weder die Anzahl der Zustände, noch die Anzahl der Transitionen in der Praxis endlich sind, ihre Darstellung dagegen endlich sein muß. In den Methoden der Programmiertechnik ist es üblich, endliche Transitionsgraphen zur Darstellung des Verhaltens einer Komponente zu verwenden.
Da der Zustandsraum einer Komponente im allgemeinen nicht endlich ist, repräsentiert ein Knoten des Graphen eine Äquivalenzklasse von Zuständen. Abbildung 2.3 zeigt das Beispiel eines einfachen Timers. Der Zustand ’waiting’ umfaßt eine nicht beschränkte Äquivalenzklasse von Datenzuständen, in denen das Attribut `counter` einen Wert besitzt, der größer als Null ist. Um detaillierter über den Zustand sprechen zu können, werden Prädikate und Pattern genutzt, die eine Partition des Zustandsraumes bilden.

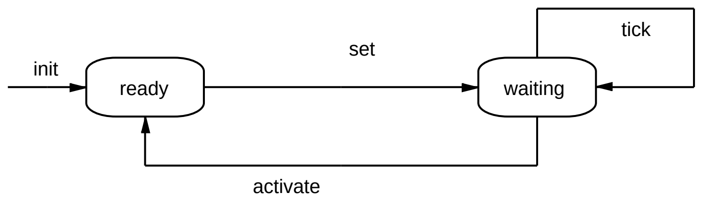


Abbildung 2.3.: Beispiel: Einfacher Timer

State	State Pattern	State Predicate
ready		counter=0
waiting		counter!=0

Tabelle 2.1.: Beispiel: Zustandstabelle

Dazu wird eine nicht weiter strukturierte Menge von Namen für Automatenzustände $\mathcal{AUT}$ eingeführt. Die endliche Menge von Automatenzuständen wird mit Hilfe einer Tabelle mit den Datenzuständen des beschriebenen Agenten verbunden. Diese Tabelle (im folgenden als *Zustandstabelle* bezeichnet) ordnet jedem Automatenzustand ein Prädikat und ein Pattern für jedes Attribut zu (siehe auch Tab. 2.1). Im Diagramm wird ein Zustand als Rechteck mit abgerundeten Ecken dargestellt, das mit dem Namen des jeweiligen Zustandes beschriftet ist (Abb. 2.3).

Transition	Input	Precond.	Output	Postcondition	Initial
set	Set(n)	n>0	[]	counter‘=n	false
activate	Tick()	counter=1	[Act(target)]	counter‘=0	false
tick	Tick()	counter>1	[]	counter‘=counter-1	false
init			[]	counter‘=0	true

Tabelle 2.2.: Beispiel: Transitionstabelle

Für die Notation von Eingabenachrichten wird ebenfalls ein Pattern, für die der Ausgabenachrichten eine Liste von Werteausdrücken verwendet. Zur genaueren Definition dienen hier ebenfalls Prädikate, die Transitionen, wo notwendig, mit Vor- und Nachbedingungen in einer adäquaten formalen Sprache, die, ähnlich des Hoare'schen Spezifikationsstils, vor beziehungsweise nach den Ein-/Ausgaben stehen, versehen (siehe auch Tab. 2.2). Transitionen werden im Diagramm als gerichtete Kanten dargestellt, die Quell- und Zielzustand miteinander verbinden (Abb. 2.3).
Initialelemente verarbeiten keine Eingabenachricht, besitzen aber ebenfalls eine Liste von Wertausdrücken als Ausgabe, die durch ein Prädikat näher definiert werden kann. Initialelemente werden im Diagramm ebenfalls durch eine gerichtete Kante dargestellt, die im Unterschied zu Transitionen jedoch nur einen Ziel- und keinen Quellzustand besitzt.
Um die graphische Darstellung des Automaten nicht zu überladen, werden Transitionen und Initialelemente mit einem Namen beschriftet, der Eingabepattern, Ausgabe und Prädikate abkürzt und diese in einer eigenen Tabelle definiert, die im folgenden kurz *Transitionstabelle* genannt wird. Dadurch wird auch Mehrfachverwendung der Tabelleneinträge möglich.
Die gerade beschriebene Darstellungsform ist allgemein genug, um Unterspezifikation zuzulassen, da Zielzustände von Transitionen prädikativ beschrieben werden, d.h. eine ganze Äquivalenzklasse von Datenzuständen als Ziel einer Transition angegeben werden kann.
Um einen abgeschlossenen und somit prüfbaren Kontext herzustellen, werden natürlich noch weitere Daten über die zu beschreibende Klasse benötigt, z.B. deren Attribute, definierte Datentypen, Ein- und Ausgabenachrichten, die eigentlich aus anderen Dokumentarten und damit anderen Werkzeugen bezogen werden.

TYPES:
ATTRIBUTES:
counter :: Int
IN:
Tick()
Set(n::Int)
OUT:
Act(target::Object)

Tabelle 2.3.: Beispiel: Deklarationen

Da solche Werkzeuge derzeit nicht verfügbar sind, wurde als Übergangslösung eine einfache Eingabemöglichkeit für extern bezogene Daten geschaffen. Attribute und Methoden können nach dem in Tabelle 2.3 abgebildeten Schema eingegeben werden.

2.3.2.1. Spezielle Attribute und Datentypen

Das standardmäßig vorhandene Attribut `actees` enthält die Menge aller Identifikatoren, die von einem Agenten aus aktiviert werden können. Um dieses Attribut in den Bedingungen verwenden zu können, wird der Datentyp `ACTEES` eingeführt, der den Typ von `actees` darstellt und für keine andere Variable verwendet werden darf.
Für die Verwendung in Prädikaten werden die Ein- und Ausgabe ebenfalls durch Variablen (`inp :: IN, out ::[OUT]`) benannt und entsprechend typisiert.
Um dabei die Erweiterung der Eingabemenge sicherzustellen, werden sogenannte erweiterbare Datentypdefinitionen für `inp` und `out` eingeführt. Eine erweiterbare Datentypdefinition kann als unvollständig in dem Sinn betrachtet werden, daß nicht alle Konstruktorterme angegeben sind. Syntaktisch wird die erweiterbare Datentypdefinition unter Benutzung von Punkten .. dargestellt, z.B. `data IN = a | b | ..`

2.3.2.2. FriscoF – Eine funktionale Sprache

Zur Angabe von Zustandspattern, Prädikaten, Vor- und Nachbedingungen innerhalb des Automatendokumentes wird eine funktionale Sprache verwendet. In [33] war keine bestimmte Syntax vorgegeben, sondern nur allgemeine Anforderungen gestellt. In dieser Arbeit wird davon ausgegangen, daß die Sprache FriscoF [15] verwendet wird. Die Implementierung von FriscoF ist in der momentanen Form sicher nicht perfekt für den gedachten Anwendungszweck geeignet, hat aber gegenüber allen anderen Sprachen den Vorteil, daß ein in Java geschriebener Parser bereits vorliegt. Für Demonstrationszwecke ist FriscoF auch in der jetzigen Form vollkommen ausreichend.

FriscoF ist stark an die Sprache Gofer angelehnt, weshalb hier auch nur die Elemente kurz erläutert werden, deren Verwendung innerhalb von Automatendokumenten angebracht ist. Gegenüber Gofer wurde die Sprache etwas vereinfacht und mit einer strikten Semantik versehen (z.B. keine unendlichen Listen). Dafür wurde die Sprache um prädikatenlogische Ausdrücke erweitert, was den Einsatz in Automatendokumenten erst möglich macht. Eine genaue und vollständige Sprachbeschreibung kann in [15] Kapitel 3 auf Seite 15 nachgeschlagen werden.

Bezeichner

Bezeichner beginnen mit einem Buchstaben und dürfen neben Ziffern ein Apostroph(') oder einen Unterstrich(_) enthalten. Es wird zwischen Groß- und Kleinschreibung unterschieden. Bezeichner für Variablen und Funktionen (varid) beginnen mit einem kleinen Buchstaben, Bezeichner für Konstruktoren (conid) mit einem großen.
Tabelle 2.4 enthält eine Auflistung aller reservierten Wörter, darunter befinden sich auch die ASCII-Ersatzzeichen für Symbole, die in Tabelle 2.5 aufgelistet sind.

ax	case	data	else	if	in	infix	infixl
infixr	let	of	op	then	type	where	ALL
ALLB	ALLP	AND	BOT	DEF	EX	EXP	FF
NOT	OR	TT	:	!	#	$	%
&	*	+	.	/	<	=	>
?	@	\	|	-	^	::	..
<-	->	=>	<=>				

Tabelle 2.4.: Reservierte Wörter, Operatorzeichen und Operatoren (FriscoF)

Graphisch	ASCII-Ersatzzeichen
$\forall$	ALL
$\forall^{\perp}$	ALLB
$\forall^{P}$	ALLP
$\exists$	EX
$\exists^{\perp}$	EXB
$\exists^{P}$	EXP
$\Rightarrow$	=>
$\Leftrightarrow$	<=>
$\wedge$	AND
$\vee$	OR
$\neg$	NOT
δ	DEF
$\perp$	BOT

Tabelle 2.5.: ASCII-Ersatzzeichen für Symbole

Standard-Datentypen

Als Standard-Datentypen existieren `Bool`, `Float`, `Int` und `Char`. `String` ist ein Typsynonym für eine Liste von `Char`.

Listen

Ist t ein Typ, so ist $[t]$ ebenfalls ein Typ und repräsentiert eine endliche Liste. Eine Liste kann mittels durch Komma (,) getrennter Aufzählung ihrer Elemente erzeugt werden. Die leere Liste wird als `[]` geschrieben. Der Operator : erzeugt eine Liste aus einem Element und und einer weiteren Liste.

Deklarationen

Variablendeklaration: $var = rhs$

Der zugewiesene Wert kann ein beliebiger Ausdruck sein, also beispielsweise eine Zahl, ein String oder eine Funktion.

Patterndeklaration $pat = rhs$

Die Patterndeklaration ist prinzipiell mit der Variablendeklaration vergleichbar, nur daß zum Beispiel in einer Patterndeklaration auch mehrere Variablen deklariert werden können. Der Wert der neu eingeführten Variablen wird ermittelt, indem das Pattern auf der linken Seite mit dem Wert auf der rechten Seite verglichen wird.

Typdeklaration $var_1, var_2, \ldots, var_n :: type$

Obwohl die explizite Angabe von Typen nicht notwendig ist, kann für jede Variable und Funktion, die durch eine Wertevereinbarung eingeführt worden ist, die Signatur angegeben werden.

Benutzerdefinierte Datentypen

Zusätzlich zu den fest vorgegebenen Datentypen können neue definiert werden. Die allgemeine Form ist:

$$\texttt{data}\ Datatype\ a_1 \ldots a_n = constr_1 | \ldots | constr_n$$

Dabei ist *Datatype* der Name des neuen Datentyps der Stelligkeit $n \geq 0$ mit den paarweise verschiedenen Typvariablen $a_1 \ldots a_n$, den Parametern des Datentyps. Der Name darf vorher nicht verwendet worden sein und beginnt mit einem großen Buchstaben. Die Konstruktorfunktionen $constr_1, \ldots, constr_n$ beschreiben, wie neue Elemente dieses Datentyps erzeugt werden, in der Regel durch $Name\ type_1 \ldots type_m$. Zwei Punkte (..) können an erster oder letzter Position anstelle einer Konstruktorfunktion verwendet werden, um anzuzeigen, daß eine Erweiterung der Definition stattfindet, bzw. stattfinden kann. Die Definition eines rekursiven Datentyps ist möglich.

Typsynonyme

Typsynonyme werden dafür verwendet, bequeme Abkürzungen oder aussagekräftige Namen für Typausdrücke verwenden zu können. Die allgemeine Form einer

Typsynonymdefinition ist:

$$\texttt{type}\ \mathit{Synonym}\ a_1 \dots a_n = \mathit{type}$$

Dabei ist *Synonym* der Name des neuen Typsynonyms der Stelligkeit $n \geq 0$ mit den paarweise verschiedenen Typvariablen $a_1 \dots a_n$, den Parametern des Typsynonyms. Der Name darf vorher nicht verwendet worden sein und beginnt mit einem großen Buchstaben. Der Typ auf der rechten Seite ist die Expansion des Typsynonyms. Werden in der Expansion Typvariablen verwendet, so müssen diese in der Typsynonym-Definition vorkommen.

Axiome

Die Formulierung von Axiomen ist in FriscoF mit der `ax`-Definition möglich. Hierzu werden die Axiome in geschweifte Klammern eingeschlossen und durch Semikolon getrennt. Zudem ist die Angabe mehrerer $\forall$- oder $\forall^{\perp}$-Quantoren möglich. Die Axiome selbst sind prädikatenlogische Ausdrücke, die optional mit einem Namen versehen werden können. Der Axiomenname steht links vom Ausdruck und wird von diesem durch einen Punkt getrennt.
Mit der `op`-Definition können zusätzlich Operatoren und Funktionen mit ihren Signaturen definiert werden, ohne eine Top-Level-Definition angeben zu müssen. Diese Spezifikationsoperatoren und -funktionen können in prädikatenlogischen Ausdrücken verwendet werden.
In den prädikatenlogischen Ausdrücken können Quantoren und Junktoren verwendet werden. Neben den üblichen Quantoren $\forall$ und $\exists$ gibt es noch die Varianten $\forall^{\perp}$ und $\exists^{\perp}$, bei denen die eingeführten Variablen zusätzlich den Wert $\perp$ annehmen können. Schließlich erlauben es die Quantoren $\forall^{P}$ und $\exists^{P}$, über konkrete Pattern Aussagen treffen zu können, z.B. $\forall^{P}(a:r) = ([x,4,5]).x < 2$

Kommentare

Ein Zeilenkommentar beginnt mit den zwei Zeichen $--$ und reicht bis zum Zeilenende. Ein geschachtelter Kommentar beginnt mit `{-` und endet mit `-}` und kann auch mehrerer Zeilen umfassen.

Grammatik

Zum Schluß noch ein Auszug aus der Grammatik:

<u>*T*op-Level</u>

topLevel	::=	**data** typeLhs **=** constrs
	|	**type** typeLhs **=** type
	|	decl
	|	**ax** { quantor }* **{** { [axiom] ||**;**}* **}**
typeLhs	::=	CONID { VARID } *

```
constrs      ::=  [..|] { constr |||}+ [||..]
             |    ..

constr       ::=  type CONOP type
             |    CONID {type}*

quantor      ::=  {ALL|ALLP} {pat :: type ||,}+ .

axiom        ::=  [{VARID |CONID} .] lExp
```

<u>*E*xpressions</u>

```
lExp         ::=  { ALL |EX | ALLP |EXP} {pat :: type ||,} + . lExp
             |    { ALLP |EXP} {pat = exp ||,}+ .lExp
             |    lExp { OR |AND |<=> |=> } lExp
             |    NOT lExp
             |    DEF exp
             |    exp = exp
             |    exp
             |    TT
             |    FF

exp          ::=  \ {apat}+ -> exp
             |    let decls in exp
             |    if exp then exp else exp
             |    case exp of {alts}
             |    opExp [::type]

opExp        ::=  opExp op opExp
             |    [-] { atomic } +

atomic       ::=  var
             |    conid
             |    INTEGER
             |    FLOAT
             |    CHAR
             |    STRING
             |    ({ exp ||,}* )
             |    ( exp op )
             |    ( op exp )
             |    [ list ]
```

| **BOT**
| **!** lExp

2.4. Erstellung und Transformation von Automaten

Im folgenden Abschnitt wird ein Verfahren vorgestellt, wie bei Neuerstellung bzw. Transformation eines Automatendokumentes mittels syntaxbasierter Beweisverpflichtungen deren Gültigkeit und Korrektheit gesichert werden kann. Bei der Neuerstellung eines Automatendokumentes sind fünf Kontextbedingungen einzuhalten, um die Wohlgeformtheit zu sichern. Unter Transformation wird an dieser Stelle die Spezialisierung der Information über das beschriebene System verstanden, im weiteren nur kurz Verfeinerung genannt. Verfeinerung ist die Hinzunahme von Informationen über das Verhalten von Agenten des Systems. Dazu wird ein zunächst für buchstabierende Automaten definierter mathematischer Verfeinerungskalkül auf Automatendokumente übertragen (siehe [33]). Die dadurch je nach angewandter Verfeinerung entstehenden Beweisverpflichtungen sichern einerseits die Wohlgeformtheit des Automatendokumentes, andererseits die Verfeinerungseigenschaft, d.h. die Tatsache, daß Vorgänger- und Nachfolgerdokument tatsächlich die semantisch gleiche Komponente beschreiben, nur eben mehr oder weniger stark unterspezifiziert.

2.4.1. Neuerstellung eines Automatendokumentes

Bei der Neuerstellung eines Automatendokumentes muß ein korrekter Lebenszyklus angegeben werden, der folgende Kontextbedingungen erfüllt:

KB1: Die in Automatendokumenten vorkommenden Patterns, Wert- und Prädikatenausdrücke müssen wohlgeformt sein. Insbesondere haben die Variablen `inp` und `out` feste Sorten, die sich aus den Eingabe- und Ausgabemethoden ableiten.

KB2: Die speziellen Attribute `self` und `actees` sind immer Teil der Attributmenge und müssen nicht explizit angegeben werden. `self`' darf nicht verwendet werden. Es gilt immer: `self`'=`self`.

KB3: Die Variablen `actees` und `actees`' dürfen nur in der Nachbedingung in der Form $\exists v.$`actees` $= v$^`actees`' [1] verwendet werden, wobei die Sorte von v sichert, daß dabei nur endlich viele Objektidentifikatoren aus `actees` entnommen werden.

[1] ^ ist die Konkatenation zweier Ströme

VB1: Erfüllbarkeit der Zustandsprädikate.

VB3: Schaltbefähigung der Transitionen. Transitionen dürfen keine falsifizierenden Nachbedingungen haben, d.h. für jedes Paar aus Quellzustand und Eingabe, das die Vorbedingung erfüllt, muß ein Zielzustand und eine Ausgabe existieren, welche die Nachbedingung erfüllen.

Die Bedingungen VB1 und VB3[2] sind auf Basis semantischer Eigenschaften formuliert und daher nicht syntaktisch überprüfbar. Es können jedoch äquivalente, auf syntaktischer Ebene formulierte Bedingungen angegeben werden, die für die Einhaltung der semantischen Bedingungen hinreichend sind. Sie werden in Form von Beweisverpflichtungen angegeben. Eine Beweisverpflichtung ist eine prädikatenlogische Formel, deren Gültigkeit für die Korrektheit eines Automatendokumentes notwendig ist. Eine Auflistung aller Beweisverpflichtungen, die hinreichend die Gültigkeit von VB1 und VB3 sichern, findet sich in [33] auf Seite 150.

2.4.2. Verfeinerung von Automatendokumenten

In [33] werden acht verschiedene Möglichkeiten angegeben, ein Automatendokument zu verfeinern, d.h. den Detaillierungsgrad der Spezifikation zu erhöhen. Diese Möglichkeiten werden fortan *(Verfeinerungs-)Regeln* genannt. Jede Regel wird über einen Kurznamen identifiziert. Es gibt folgende acht Regeln, die im Anschluß ausführlich vorgestellt werden:

addS: Erweiterung der Menge der Automatenzustände.

remS: Einschränkung der Menge der Automatenzustände, wenn die entfernten Zustände nicht erreichbar sind.

refS : Verfeinerung von Automatenzuständen.

addT: Hinzunahme von Transitionen für Kombinationen aus Eingabe und Quellzustand, für die bisher keine Transition existiert hat (Totalisierung).

remT: Entfernung von Transitionen des Diagramms, wenn Alternativen existieren (Entfernung von Unterspezifikation).

refT : Verfeinerung einer Transition.

remI : Entfernung von Initialelementen (Entfernung von Unterspezifikation).

refI : Verfeinerung von Initialelementen.

[2]VB2 entfällt aufgrund von geringfügigen Modifikationen (siehe 2.5).

2.4.2.1. Erweiterung der Menge der Automatenzustände

Es können jederzeit neue Automatenzustände hinzugefügt werden. Ein neuer Automatenzustand kann zum Beispiel bisher nicht in anderen Automatenzuständen enthaltenen Datenzustände übernehmen. Zu jedem neuen Automatenzustand kann ein passendes Zustandsprädikat angegeben werden, das die zugehörige Datenzustandsmenge beschreibt. Durch die Erweiterung des Automaten um neue Zustände wird der Automat in den neuen Zuständen partiell. Auch sind die neu eingeführten Zustände zunächst nicht erreichbar. Durch Hinzufügen von Transitionen, ausgehend von bisher erreichbaren Zuständen, werden die neuen Zustände ebenfalls erreichbar und tragen zur Semantikbildung bei. Um die Kontextbedingungen zu sichern, entstehen Beweisverpflichtungen dergestalt, daß keinem der neuen Automatenzustände eine leere Menge von Datenzuständen zugeordnet wird, d.h. alle Zustandsprädikate müssen erfüllbar sein.

2.4.2.2. Einschränkung der Menge der Automatenzustände

Eine Menge von Automatenzuständen kann entfernt werden, wenn die zu entfernenden Zustände im Transitionsdiagramm nicht erreichbar sind, d.h. die zu entfernenden Zustände sind keine Initialzustände und es gibt keine Transition, die aus der Menge der verbleibenden Zustände in die Menge der zu entfernenden Zustände führt. Die Automatenzustände tragen in diesem Fall nicht zum Verhalten des beschriebenen Objekts bei. Natürlich werden alle Transitionen, deren Quellzustand gelöscht wird, automatisch mit entfernt. Analog zur Erweiterung der Automatenzustände ist deren Einschränkung auch keine echte Verfeinerung. Beweisverpflichtungen entstehen an dieser Stelle keine, allerdings müssen die angegebenen Beschränkungen eingehalten werden.

2.4.2.3. Hinzunahme von Transitionen des Diagramms

Ist die Transitionsrelation eines Automaten partiell, weil es für eine Kombination aus Quellzustand und Eingabenachricht keine Transition gibt, die schalten kann, wird damit Chaos modelliert. Für eine robuste Verfeinerung von Chaos können Transitionen in das Diagramm aufgenommen werden, die ein Verhalten für unterspezifizierte Paare aus Quellzustand und Eingabenachricht definieren. Für die Einhaltung der Kontextbedingungen müssen die Schaltbereiche der neuen Transitionen disjunkt zu den Schaltbereichen bereits existierender Transitionen sein. Es entstehen folgende Beweisverpflichtungen für jede neu hinzugefügte Transition:

- Die Transition muß schaltbar sein, d.h. wenn das Paar aus Quellzustand und Eingabe die Vorbedingung erfüllt, muß ein Zielzustand und eine Ausgabe existieren, welche die Nachbedingung erfüllen.

- Die Transition muß disjunkt gegenüber den alten Transitionen sein, d.h. es darf keine Belegung für Quellzustand und Eingabenachricht geben, welche die Vorbedingungen einer neuen und einer alten Transition erfüllen, d.h. daß beide Transitionen schalten können.

Zu beachten ist, daß die neuen Transitionen in einem Schritt gemeinsam hinzugefügt werden müssen, wenn sie sich überlappen sollten.
Beweisverpflichtungen zur Disjunktheit der Schaltbereiche entstehen nur für je ein Paar einer neuen und einer alten Transition mit demselben Quellzustand.
In der Softwaretechnik kann diese Verfeinerungstechnik zum Beispiel dazu benutzt werden, um Partialitäten eines Automaten als Fehlerfälle robust zu implementieren.

2.4.2.4. Entfernung von Transitionen eines Diagramms

Gibt es für eine gegebene Eingabenachricht und einen gegebenen Datenzustand mehrere schaltbereite Transitionen, so kann eine derartige Unterspezifikation durch die Entfernung einer der beteiligten Transitionen vermindert werden. Eine Transition des Automatendokumentes kann also entfernt werden, wenn ihr Schaltbereich vollständig von anderen Transitionen überdeckt ist. Es entsteht eine Beweisverpflichtung, die diesen Umstand sichert.

2.4.2.5. Verfeinerung von Transitionen

Ein weiterer möglicher Schritt zur Verfeinerung von Automatendokumenten ist die Verfeinerung von Transitionen. Dazu wird eine neue Transition hinzugefügt, die eine Spezialisierung einer bereits vorhandenen Transition ist. Es entstehen folgende Beweisverpflichtungen:

- Die neue Transistion muß schaltbar sein.
- Die Transition muß eine Verfeinerung sein, d.h. für jede Belegung von Quellzustand und Eingabe, für welche die neue Transition schalten kann, gibt es eine alte Transition, die ebenfalls schalten kann, und der zugehörige Zielbereich der neuen Transition muß eine Spezialisierung der neuen Transition sein.

Die Hinzunahme einer verfeinerten Transition ist eine semantikerhaltende Transformation, denn die Hinzunahme einer neuen Transition im Automatendokument, die bereits durch eine andere Transition abgedeckt wird, verändert kein Verhalten. Interessant wird dieser Schritt vor allem in Kombination mit der Entfernung der verfeinerten Transition. Das kann beispielsweise dazu benutzt werden, um Bedingungen einer Transition neu zu formulieren, um sie in eine ausführbare Fassung zu bringen. Genauso können Nachbedingungen oder Ausgabeausdrücke spezialisiert werden. Durch wiederholte Einführung solcher Transitionen, die jeweils einen Teil des Schaltbereiches einer Transition abdecken, kann eine Transition durch mehrere Transitionen verfeinert werden, aber auch die Zusammenlegung von Transitionen mit demselben Quell- und Zielzustand ist möglich.

2.4.2.6. Einschränkug der Menge der Initialelemente

Besitzt ein Automatendokument mehrere Initialelemente, so können Initialelemente entfernt werden, um damit die initiale Ausgabe und den Initialzustand präziser festzulegen, also die Spezifikation zu verfeinern. Dabei muß darauf geachtet werden, daß mindestens ein erfüllbares Initialelement im Automatendokument erhalten bleibt, da das Dokument sonst eine leere Menge von Verhalten beschreibt. An dieser Stelle entstehen keine weiteren Beweisverpflichtungen.

2.4.2.7. Verfeinerung von Initialelementen

Analog zur Verfeinerung von Transitionen können auch Initialelemente verfeinert werden. Dabei wird ein neues Initialelement hinzugefügt, das eine Spezialisierung bereits vorhandener Initialelemente ist. Diese Transformation ist keine Verfeinerung im engeren Sinne. Wie die Verfeinerung von Transitionen wird sie vorbereitend für weitere echte Verfeinerungsschritte eingesetzt. Es entstehen Beweisverpflichtungen, um zu zeigen, daß das neue Initialelement eine Spezialisierung eines vorhandenen Initialelementes darstellt.

2.4.2.8. Teilung von Automatenzuständen

Um eine Verhaltensbeschreibung zu präzisieren, ist es oft sinnvoll, einen Automatenzustand zu teilen und das Verhalten für jeden neuen Automatenzustand individuell zu spezialisieren. Die jeweils zugeordneten Datenzustandsräume partitionieren den Datenzustandsraum des alten Automatenzustandes. Bei der Teilung von Automatenzuständen ist es notwendig, beteiligte Transitionen und Initialelemente zu modifizieren. Dadurch wird diese Verfeinerung komplex. Die betroffenen Transitionen und Initialelemente werden aufgespalten und dabei vervielfältigt. Es entstehen folgende Beweisverpflichtungen:

- Die neuen Zustände müssen erfüllbar sein.

- Es muß sichergestellt werden, daß die neuen Zustände den alten Zustand tatsächlich vollständig partitionieren.

Die Beweisverpflichtungen sichern aber auch, daß die Transformation tatsächlich eine Verfeinerung des Zustandsraumes darstellt. Die Schaltbarkeitsbedingung der betroffenen Transitionen wird dadurch gesichert, daß die Vorbedingungsprädikate der betroffenen Transitionen dergestalt eingeschränkt werden, daß die Schaltbarkeit weiterhin ausschließlich von der Vorbedingung festgelegt wird. Dabei werden manche dieser Vorbedingungen zu *False* reduziert, und die zugehörigen Transitionen können entfernt werden.

2.5. Modifikationen der Methodik

Wie in Abschnitt 2.1 erwähnt, basiert die in diesem Kapitel dargestellte Theorie auf der in [33] beschriebenen Methodik. Es wurden jedoch einige kleinere Modifikationen vorgenommen, welche durch die leicht geänderte Semantik der Automatendokumente begründet sind und im folgenden gesammelt dargestellt werden:

- Aufgabe der Disjunktheit der Datenzustände:
  Die Disjunktheit von Datenzuständen ist im Gegensatz zu [33] durch einen implizit mitgeführten Kontrollzustand, der auf den disjunkten Bezeichnern für Zustände basiert, automatisch gegeben und braucht deshalb nicht als Kontextbedingung gefordert werden.

  Die Disjunktheit der Datenzustände ist damit automatisch ohne Angabe von Beweisverpflichtungen gesichert, indem gefordert wird, daß die Bezeichner eindeutig sind. Somit kann die Menge der Beweisverpflichtungen verringert und auch die Handhabung vereinfacht werden, da die in vielen Anwendungen häufig zur besseren Strukturierung benutzten reinen Kontrollzustände jetzt als Zustände ohne Zustandspattern und Prädikat dargestellt werden können.

  Durch die Einführung impliziter Kontrollzustände muß darauf geachtet werden, daß diese im Gegensatz zu [33] jetzt in den generierten Beweisverpflichtungen berücksichtigt werden müssen, was allerdings automatisch durch das Werkzeug realisiert werden kann und zu der erwähnten Reduzierung der Anzahl Beweisverpflichtungen führt.

- Als Notation für die Angabe von Ausgabemethoden wird statt

  $$\text{obj.method}(\text{param}_1,\ldots,\text{param}_n)$$

  jetzt

  $$\text{method}(\text{obj},\text{param}_1,\ldots,\text{param}_n)$$

  verwendet.

  Damit wird einerseits die Kompatibilität zu existierenden funktionalen Sprachen erhöht, die nicht objektorientiert erweitert wurden, andererseits wird damit gegenüber [33] wieder deutlicher herausgestellt, daß die verwendete Methodik keineswegs auf die Entwicklung objektorientierter Software beschränkt ist, sondern sich auch generell für den Systementwurf eignet.

- Die Unterstützung der speziellen Attribute `self` und `actees` zur Darstellung des Identifikators eines Objekts beziehungsweise der von ihm erzeugbaren Abkömmlinge ist optional und hängt ausschließlich von der konkret verwendeten funktionalen Sprache ab. Die in dieser Arbeit zu Demonstrationszwecken verwendete Sprache FriscoF unterstützt die beiden Attribute in der gegenwärtigen Version nicht. Diese sind daher gegebenenfalls explizit anzugeben.

- Die spezielle Variable heißt jetzt `inp` anstelle von `in`, um Namenskonflikte mit Schlüsselwörtern in vielen funktionalen Sprachen zu vermeiden.

- Die in [33] explizit genannten fünf Kontextbedingungen wurden durch eine Reihe von Bedingungen ergänzt, die sich vor allem aus der pragmatischen Umsetzung ergeben:

  – Zustandsbezeichner müssen global eindeutig sein.
  – Transitionsbezeichner dürfen mehrfach verwendet werden, wenn sich die zugehörigen Transitionen genau in der Kombination aus Quell und Zielzustand unterscheiden. Dadurch wird die Mehrfachverwendung von Transitionen realisiert. Alle anderen Transitionsbezeichner müssen global eindeutig sein.
  – Quell und Zielzustand einer Transition müssen definiert sein.
  – Ein Initialelement darf keinen Quellzustand besitzen. Ein Zielzustand muß vorhanden sein.
  – Es muß stets mindestens ein Initialelement vorhanden sein (auch bei Beginn der Verfeinerung).
  – Zustands und Transitionsbezeichner dürfen nicht leer sein.
  – Das Automatendokument muß über einen nicht leeren Bezeichner einer Klasse zugeordnet werden.
  – Das Automatendokument muß vom Typ `class` oder `type` sein.

2.6. Zusammenfassung

Ein Automatendokument beschreibt mögliche Zustandsübergänge von Elementen einer Klasse bei der Verarbeitung von Nachrichten und die Reaktion darauf.
Die Semantik des Automatendokuments wird durch Rückführung auf die Semantik von buchstabierenden Automaten definiert, wodurch eine zweistufige Semantikdefinition für Automatendokumente vorgenommen wird. Zunächst werden diese in einen buchstabierenden Automaten um- und dann über die formale Fundierung mit dem Systemmodell in Beziehung gesetzt.
Jeder in einem Automatendokument enthaltene Automat ist einer Klasse zugeordnet. Er beschreibt das Verhalten der zu dieser Klasse gehörenden Agenten.

In der konkreten Syntax wird der potentiell unendliche Zustandsraum durch endlich viele Äquivalenzklassen unterteilt. Zustände und Transitionen erhalten Bezeichner, so daß die graphische Notation in Beziehung zu einer Zustands- bzw. Transitionstabelle gesetzt werden kann, in der dann beispielsweise mittels Prädikaten eine detailliertere Aussage über Zustände und Transitionen möglich ist.

3. Von der Theorie zur Praxis

Dieses Kapitel beschäftigt sich ausschließlich mit der praktischen Verwendung des STDA-Werkzeugs. Die dafür notwendigen theoretischen Grundlagen wurden in Kapitel 2 ausführlich erläutert.
Die folgenden Abschnitte sind thematisch geordnet, so daß sie in etwa einem typischen Arbeitsablauf entsprechen und sind jeweils zweigeteilt. Der erste Teil beschreibt alle Bedienungselemente und Funktionen zu einem bestimmten Arbeitsschritt. Der zweite Teil enthält ein Tutorial, in dem das Wissen aus dem ersten Teil anhand eines abschnittsübergreifenden Beispiels eingeübt wird. Zur besseren Unterscheidung ist der zum Tutorial gehörende Text leicht hervorgehoben.
Im folgenden werden zunächst alle Schritte erklärt, die notwendig sind, bevor mit der eigentlichen Arbeit begonnen werden kann.
Danach wird erklärt, wie ein neues Automatendokument erstellt wird. Der letzte Abschnitt beschäftigt sich mit der Modifikation eines bestehenden Dokumentes (Verfeinerung).

3.1. Arbeiten mit dem OEF

Das OEF ist die für das STDA-Werkzeug notwendige Laufzeitumgebung. Es verwaltet mehrere Werkzeuge und die zugehörigen Daten in Form von Dokumenten, die wiederum in Projekten gespeichert sind. Eine vollständige Beschreibung aller Bedienungselemente findet sich in [30], an dieser Stelle werden nur die für das STDA-Werkzeug wichtigen Elemente ausschnittsweise beschrieben.
Nach dem Login-Dialog erscheint der Projektmanager. In diesem Dialog können Dokumente in vorhandenen Projekten gesucht, geöffnet, gelöscht, oder neu angelegt werden. Gleiches gilt für die Projekte. Ein Projekt kann jeweils weitere Projekte als Unterprojekte enthalten.
Beim Anlegen eines neuen Dokumentes hat man die Auswahl zwischen einem leeren Dokument und einer Liste von Templates. *Templates* sind vorformatierte Vorlagenmuster, die bereits vorbereitete Parts enthalten können. Ein *Part* ist eine Komponente des Dokumentes, technisch eine Einheit aus Information in einer bestimmten Repräsentation und einem Werkzeug zur Darstellung und ggf. Modifikation der Information, beispielsweise einer Tabelle, oder eines Diagramms.

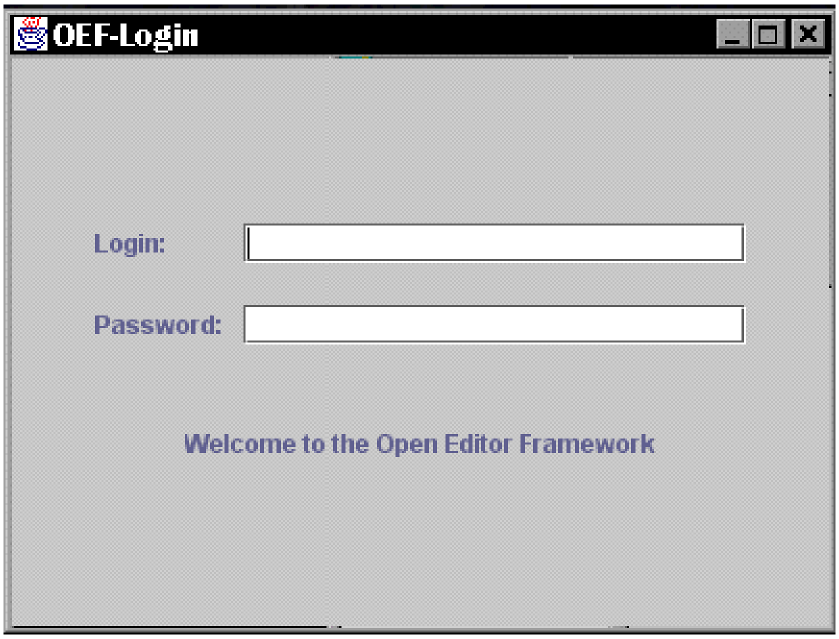


Abbildung 3.1.: Login Dialog

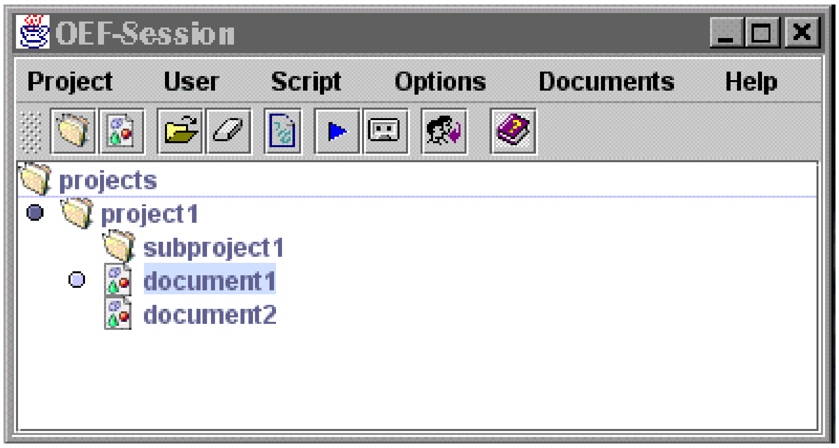


Abbildung 3.2.: Projektmanager

In den meisten Fällen ist es ausreichend, ein leeres Dokument zu erstellen.

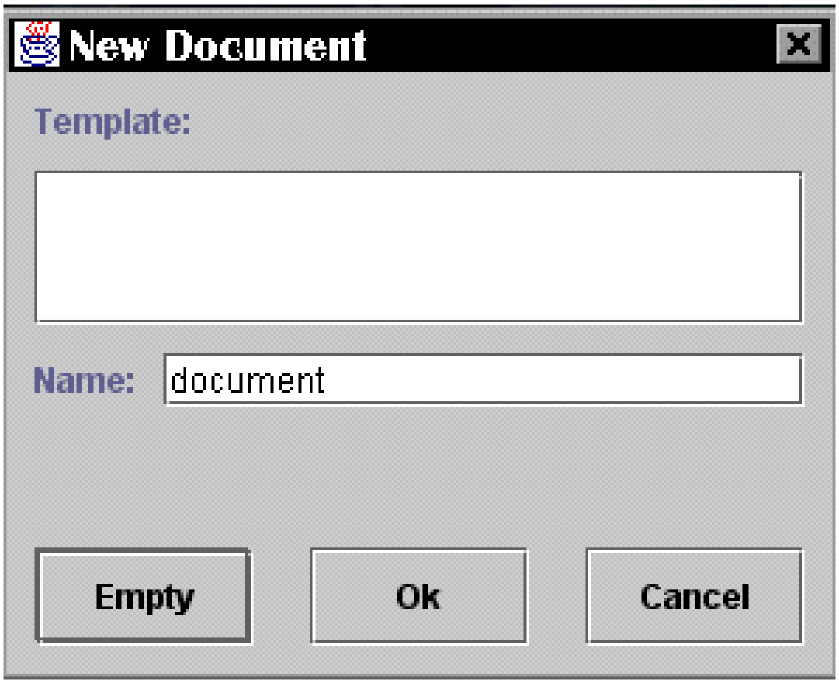


Abbildung 3.3.: Neues Dokument erstellen

Ihre Firma wurde zwei Tage nach der Vorstellung von Windows 98 in Chicago völlig überraschend mit dem Einbau einer Alarmanlage in die Skihütte von Bill Gates beauftragt. Als Mitarbeiter der Softwareabteilung wurde Ihnen die verantwortungsvolle Aufgabe übertragen, das zentrale Steuerungsmodul zu entwerfen. Die Hütte (Abb. 3.4) hat zwei Fenster und eine Türe, die mit Sensoren abgesichert werden. Ein zusätzlicher Bewegungsmelder überwacht den einzigen Raum. Auf Wunsch von Bill wird zusätzlich ein weiterer Bewegungsmelder an der Außenseite der Hütte installiert, um die Annäherung aufdringlicher Journalisten und feiger Tortenwerfer über einen stillen Alarm zu melden. Neben der Alarmsirene existiert also ein Blinklicht im Inneren der Hütte, das unerwünschte Besucher meldet. Die Alarmanlage wird über einen kleinen Handsender bedient. Der zugehörige Empfänger ersetzt den veralteten Schlüsselschalter. Über eine LED-Zeile (Abb. 3.12) am Zentralmodul läßt sich jederzeit der Zustand der Anlage und evt. Störungen ablesen. Die Alarmanlage besteht aus bewährten modularen Komponenten, die von einer programmierbaren Zentrale aus gesteuert werden. Die Zentraleinheit kann mittels einer objektorientierten Sprache programmiert werden. Es existieren bewährte Klassen zur Ansteuerung jedes eingesetzten Hardwaremoduls. Mit der Klasse für die zentrale Steuerung gab es in letzter Zeit jedoch häufig Probleme, weshalb entschieden wurde, diese Klasse völlig neu, von Grund auf zu entwerfen und dabei die neuesten zur Verfügung stehenden Werkzeuge einzuset-

Abbildung 3.4.: Die Skihütte von Bill Gates

zen. Ihr Projekt ist also zusätzlich ein Eignungstest für die kürzlich erworbene Werkzeugplattform OEF.
Sie beginnen zunächst mit einem einfachen Modell einer Alarmanlage. Die Feinheiten müßten sich später einfügen lassen. Der Zustand der Alarmzentrale kann mit drei einfachen Wörtern beschrieben werden:

Unscharf: Quasi ein Wartezustand. Von hier aus kann die Anlage nur scharfgestellt werden.

Scharf : Die Anlage ist aktiviert. Die Auslösung eines Sensors führt zu einem Alarm. Solange noch kein Alarm ausgelöst wurde, kann die Anlage deaktiviert werden.

Alarm : Wurde ein Sensor ausgelöst, kommt es zu einem Alarm. Die Unterscheidung zwischen stillem und lautem Alarm behalten Sie im Hinterkopf, verschieben sie aber auf einen späteren Zeitpunkt.

Starten Sie zunächst die Werkzeugplattform OEF und legen Sie ein neues leeres Projekt an. Wählen Sie dazu `Project|New Project`. Im Projektmanager erscheint ein neues Projekt mit der Bezeichnung `project1`.
Geben Sie dem Projekt jetzt einen aussagekräftigen Namen. Dazu muß das Projekt selektiert werden, indem Sie den Mauszeiger über den Schriftzug `project1`

bewegen und einmal mit der linken Maustaste klicken. Sie sehen, daß der Schriftzug hervorgehoben dargestellt wird. Jetzt betätigen Sie erneut die linke Maustaste. Der Schriftzug wird von einem Eingabefeld überlagert. Ändern Sie die Bezeichnung jetzt in `alarm` *(siehe Abbildung 3.5). Abschließend betätigen Sie die Taste* `ENTER`*.*

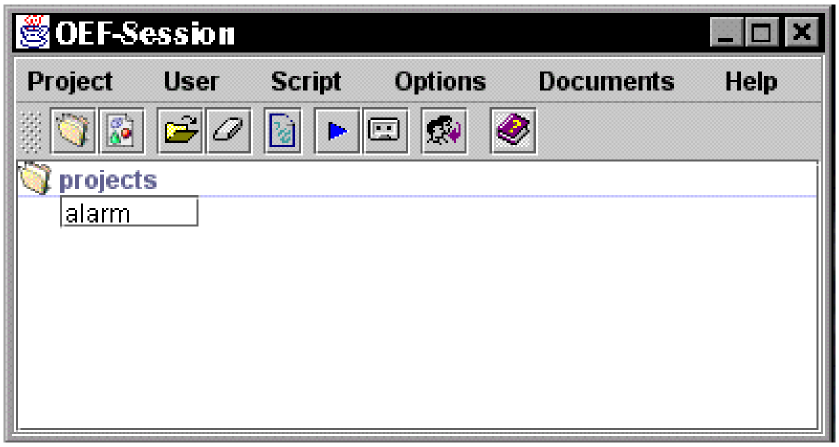


Abbildung 3.5.: Änderen des Projektnamens

Legen Sie jetzt ein neues Dokument an. Wählen Sie dazu im Projektmanager den Menüpunkt `Project|New Document`*. Nachdem Sie das Projekt* `alarm` *selektiert haben, geben Sie als Namen* `Beginn` *ein und klicken auf den Knopf* `Empty`*. Nach kurzer Wartezeit erscheint ein neues leeres Dokument auf dem Bildschirm.*

3.1.1. Der Dokumentkontroller STDA

Ein *Dokumentenkontroller* ist die Hauptkomponente eines komplexen OEF-Werkzeuges, das aus mehreren Teilwerkzeugen und Editoren besteht. In der Regel modifiziert er ein Dokument dahingehend, daß keine freie Gestaltung der Beschreibungselementabfolge mehr möglich ist, sondern, ähnlich einem Formular, nur noch vorbestimmte Informationsfelder genutzt werden können. Auf Basis dieses Eingabeformulars wird dann eine höhere Werkzeugfunktionalität implementiert, die sich auf Daten des Formulars abstützt.
Ein Dokumentenkontroller wird wie ein gewöhnlicher OEF-Part geladen, mit dem Unterschied, daß er nach dem Start die Kontrolle über das Dokument übernimmt, um beispielsweise weitere Werkzeuge zu laden.
Dokumentenkontroller unterscheiden sich von gewöhnlichen Parts durch den Namenszusatz `Assistant`.
Der konkrete Dokumentenkontroller `STD-Assistant` erzeugt bei seinem Aufruf vier weitere Eingabeeditoren (Abb. 3.8), ein Hilfswerkzeug und ein Paneel zur

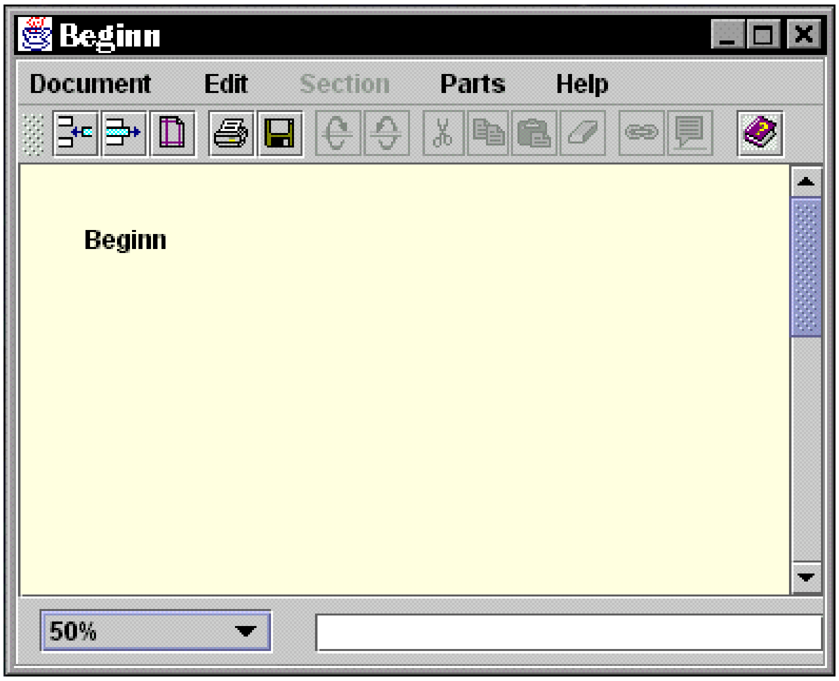


Abbildung 3.6.: Leeres Dokument

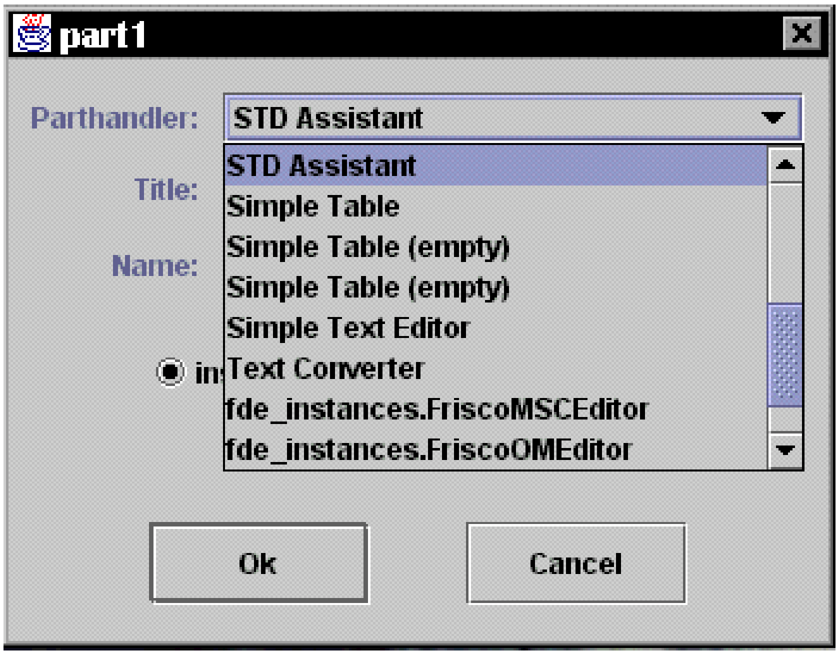


Abbildung 3.7.: Auswahl eines Parts

Steuerung der Hauptfunktionen (Abb. 3.9). Dieses Kontrollpaneel besteht von oben nach unten aus:

- Einer optischen Fehlerzustandsanzeige.

  Ist kein Fehler aufgetreten, ist diese Anzeige grau und nicht vom normalen Dialograhmen zu unterscheiden. Ist dagegen ein Fehler aufgespürt worden, blinkt diese Anzeige rot.

- Einer Reihe von Steuerelementen.

  Hiermit können die drei Hauptfunktionen des Werkzeugs aktiviert bzw. deaktiviert werden. Die Betätigung eines Knopfes aktiviert die rechts davon beschriebene Funktion für einen Durchlauf bzw. testet im Fall der Verfeinerung, ob in den Verfeinerungsmodus gewechselt werden kann. Die Markierung des Rechtecks jeweils links neben der beschriebenen Funktion aktiviert die zugehörige JIT[1] Funktion, d.h. die Funktionen werden automatisch bei jeder Änderung des Automatendokumentes ausgeführt. Es ist jedoch zu beachten, daß aus Geschwindigkeitsgründen nicht sämtliche Prüfungen im JIT-Modus ausgeführt werden können. Dies gilt nicht für den Verfeinerungsmodus. Dort werden generell alle Prüfungen durchgeführt. Weiterhin ist zu beachten, daß eine gewählte Funktion jeweils alle weiteren links von ihr stehenden Funktion automatisch mitaktiviert, da es beispielsweise keinen Sinn macht, den Kontext zu überprüfen, ohne daß vorher die Syntax überprüft wurde. Der Verfeinerungsmodus kann außerdem nicht deaktiviert werden.

- Einem Fehleranzeigefeld.

  Hier werden alle aufgetretenen Fehler aufgelistet. Es werden jeweils nur die gegenwärtigen Fehler angezeigt. Wird ein Fehler beseitigt, verschwindet auch die zugehörige Fehlermeldung.

  Wird ein Fehlereintrag selektiert, wird die Fehlerstelle, wenn möglich, farbig hervorgehoben.

- Einem Textfeld.

  Hier muß der Bezeichner der Klasse (oder des Agenten) eingetragen werden, dem dieses Automatendokument zugeordnet ist.

- Einem Auswahlfeld. Hier muß der Typ des Automatendokumentes ausgewählt werden.

[1]Just In Time

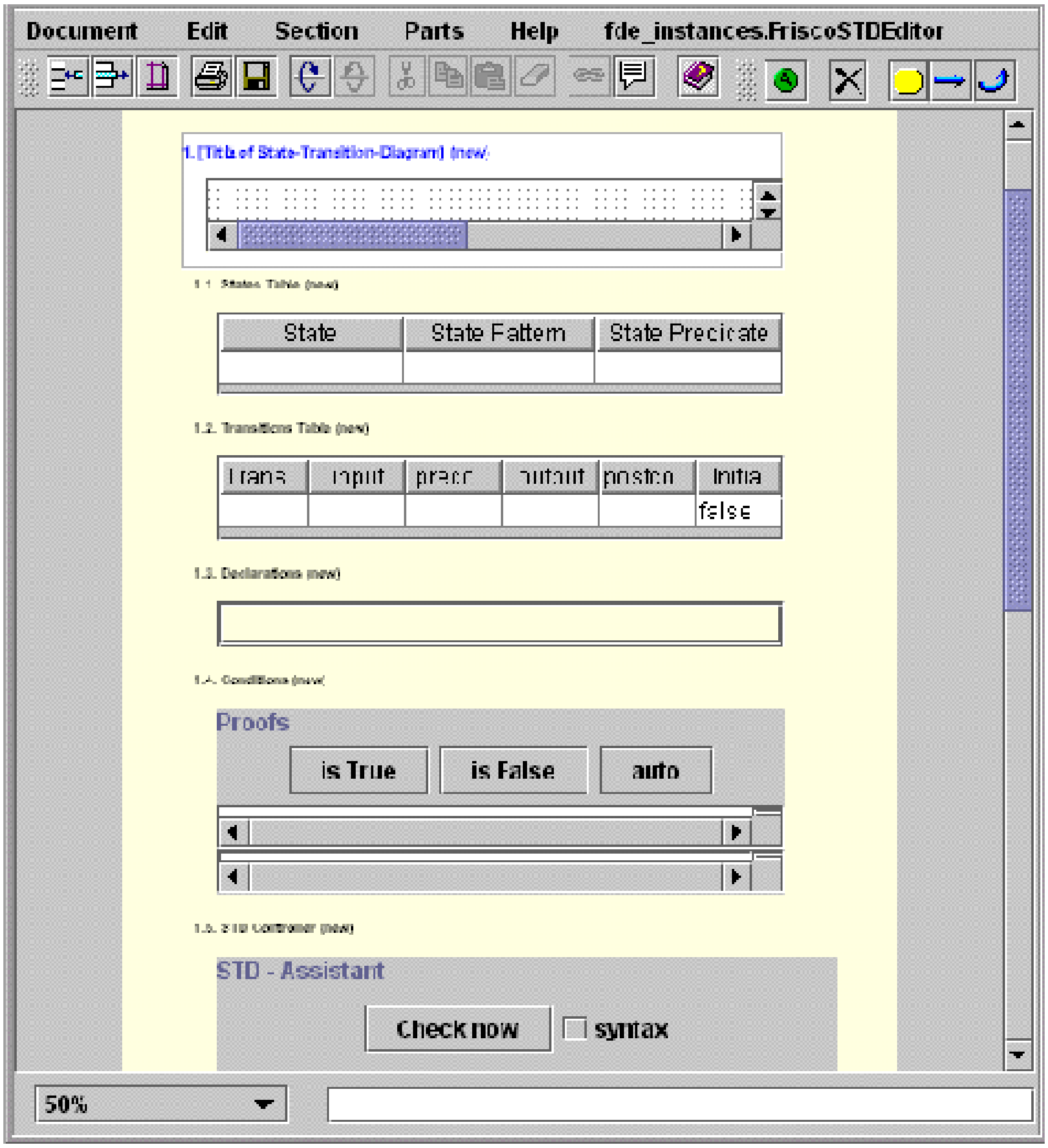


Abbildung 3.8.: Gesamtlayout eines STDA-Dokumentes

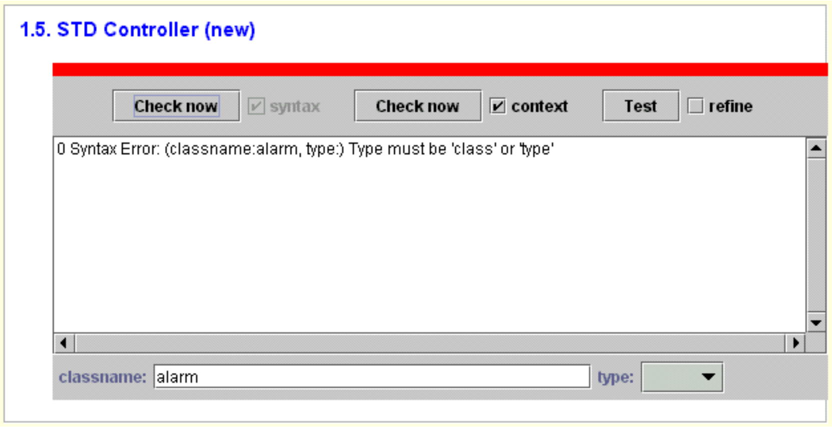


Abbildung 3.9.: Steuerpanel des Dokumentenkontrollers

Starten Sie das Werkzeug STDA. Wählen Sie dazu den Menüpunkt `Document|New Part`*. Es erscheint Ein Dialogfenster. Dort wählen Sie zunächst den Typ des einzufügenden Werkzeugs. Klicken Sie mit der Maus auf das nach unten gerichtete gefüllte Dreieck ganz rechts in der Zeile* `Parthandler`*. Es erscheint ein Listenfeld. Wählen Sie den Eintrag* `STD Assistant`*. Titel und Namen des Dokumentes können Sie nach Belieben ändern.*
Jetzt müssen Sie den Namen der Klasse angeben, die durch dieses Dokument beschrieben werden soll. Suchen Sie das Kontrolpaneel am Ende des Dokumentes auf und tragen in das Feld neben 'classname:' die Bezeichnung `alarm` *ein. Anschließend wählen Sie aus der Liste rechts neben dem Klassennamen den Typ* `class` *aus.*

3.2. Konstruktion eines Automaten

3.2.1. Zeichnen mit dem Diagrammeditor

Sämtliche Zeichenfunktionen des Diagrammeditors (das ist der Part am Anfang des Automatendokumentes) können in aller Ausführlichkeit in [6] nachgelesen werden. Hier nur die aller wichtigsten Funktionen. Ist der Diagrammeditor selektiert, wird die Werkzeugleiste um fünf Funktionen erweitert. Diese sind von rechts nach links:

- Hinzufügen einer gebogenen Transition.
- Hinzufügen einer geraden Transition.

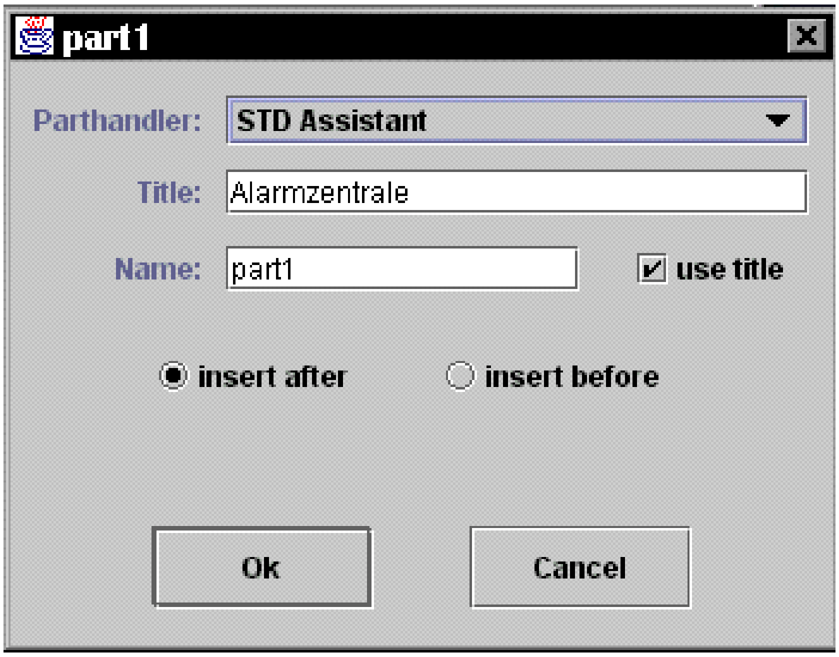


Abbildung 3.10.: Werkzeug STD-Assistant starten

- Hinzufügen eines Zustandes.
- Löschen der selektierten Elemente.
- Hinzufügen einer Annotation.

Um ein Textfeld zu editieren, muß das zugehörige Element selektiert sein. Ein SHIFT-Doppelklick wechselt in den Editiermodus. Ein Klick außerhalb der Markierung beendet das Editieren.

Um ein Textfeld relativ zum ihm zugeordneten Element zu verschieben, muß der Mauszeiger auf das Element zeigen und die Taste '+' betätigt werden. Alle anderen Elemente werden grau. Jetzt kann das Textfeld selektiert und verschoben werden. Ein Klick außerhalb des Elementes beendet diesen Modus.

Zeichnen Sie zunächst die anfänglichen drei Zustände 'Unscharf', 'Scharf' und 'Alarm' und die für jeden Zustand wichtigste Transition. Sie sind sich zunächst noch unsicher, ob die Alarmanlage nach der Initialisierung im Zustand 'Scharf', oder 'Unscharf' ist. Lassen Sie sich alle Möglichkeiten offen und zeichnen zwei Initialelemente ein (siehe auch Abb. 3.11).

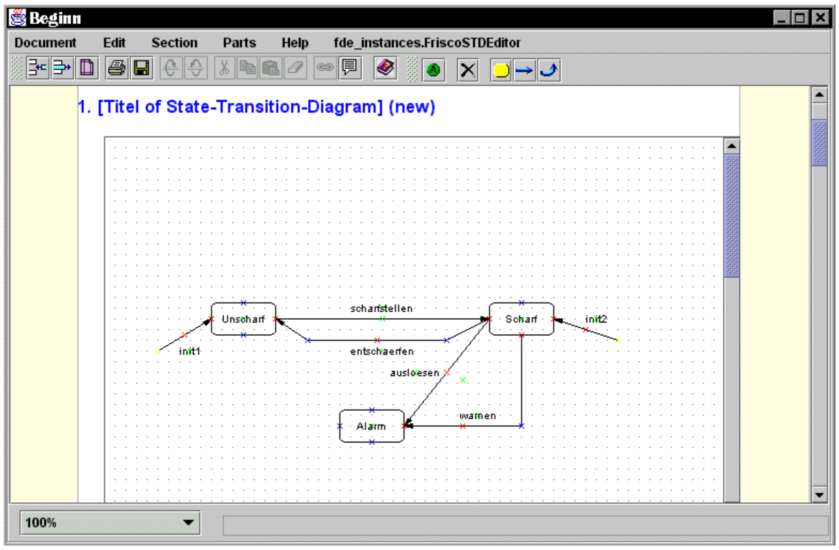


Abbildung 3.11.: Eine einfache Alarmanlage

3.2.2. Die Zustandstabelle

Die Zustandstabelle ist der Part unterhalb des Automatendiagramms. Hier können die im Diagramm mit Namen qualifizierten Zustände durch Angabe von Zustandspattern und Zustandsprädikaten detaillierter spezifiziert werden. Die Zustandstabelle hat drei Spalten. Die erste Spalte enthält die Namen der Zustände, wie sie auch im Diagramm zu sehen sind.
Die zweite Spalte enthält Zustandsprädikate. Zustandsprädikate beziehen sich immer auf ein Attribut und geben ein Wertemuster für ein bestimmtes Attribut an, das Konstanten und Variablen enthalten darf. In einem Feld dürfen mehrere Zustandspattern angegeben werden, die durch Semikolon getrennt werden müssen.
Die dritte Spalte enthält prädikatenlogische Ausdrücke, die Attribute und Variablen der Zustandspattern eines Zustandes näher beschreiben. In jedem Feld darf nur ein Ausdruck stehen, es können aber beliebige Teilausdrücke mit `AND` verknüpft werden.
Die Zustandstabelle basiert auf dem OEF-Part `SimpleTable`. Eine genaue Bedienungsanleitung ist in [30] nachzulesen. Über die Werkzeugleiste können jederzeit neue Zeilen in die Tabelle eingefügt werden. Eine Zelle wird durch einen Doppelklick editiert. Wird ein Eintrag in eine leere Zeile vorgenommen, wird automatisch ein neuer Zustand erzeugt. Sind nach einer Eingabe alle Felder einer Zeile leer, wird der Zustand, der vorher zu dieser Zeile gehörte, automatisch auch im Diagramm gelöscht. Gleiches gilt, wenn eine nicht leere Zeile ganz entfernt wird. Das Hinzufügen neuer Spalten ist nicht möglich.

Die momentan vorhandenen Zustände sind reine Kontrollzustände, d.h. außer einem implizit vorhandenen Kontrollzustand werden keine weiteren Aussagen über Attribute gemacht, weshalb in der Zustandstabelle keine weiteren Eintragungen notwendig sind.

3.2.3. Die Transitionstabelle

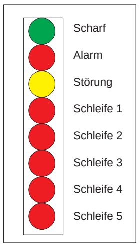


Abbildung 3.12.: LEDs

Die Transitionstabelle ist der Part unterhalb der Zustandstabelle. Hier können die im Diagramm mit Namen qualifizierten Transitionen durch Angabe von Eingabepattern, Ausgaben, Vor- und Nachbedingungen detaillierter spezifiziert werden. Die Transitionstabelle hat sechs Spalten. Die erste Spalte enthält die Namen der Transitionen, wie sie auch im Diagramm zu sehen sind.
Die zweite Spalte enthält ein Eingabepattern. Ein Eingabepattern enthält den Namen einer Eingabemethode und ein Muster der Parameter der Eingabemethode. Dabei kann es sich um Konstanten oder Variablen handeln. Es kann nur ein Eingabepattern angegeben werden. Es bestimmt zusammen mit der in der dritten Spalte angegebenen Vorbedingung, bei welcher Art von Eingabe die beschriebene Transition aktiviert wird.
Die Vorbedingung ist ein prädikatenlogischer Ausdruck, der Variablen aus dem Zustandspattern des Quellzustandes und Variablen aus dem Eingabepattern näher beschreibt, für den Fall, daß die Transition schalten kann.
Die vierte Spalte enthält eine Liste von Ausdrücken, die ausgegeben werden, falls die Transition schalten kann. Der Ausgabeausdruck darf sich neben Attributen auch auf die gleichen Variablen beziehen, wie die Vorbedingung.
Die fünfte Spalte enthält einen prädikatenlogischen Ausdruck für die Nachbedingung, in der zusätzlich Variablen des Zielzustandspatterns in gestrichener Form (um anzuzeigen, daß der Attributwert nach dem Schaltvorgang der Transition gemeint ist) verwendet werden dürfen.
Die sechste Spalte gibt an, ob es sich bei der betroffenen Transition um eine Initialtransition handelt. Es sind nur die Werte `true` und `false` erlaubt. Handelt es sich um eine Initialtransition, müssen die Felder für Eingabepattern und Vorbedingung leer bleiben, da Initialtransitionen keine Eingabe verarbeiten.
Auch die Transitionstabelle basiert auf dem OEF-Part `SimpleTable`. Alle in Abschnitt 3.2.2 beschriebenen Funktionen gelten analog. Um eine Transition zu löschen reicht es, wenn alle Felder außer dem in der letzten Spalte leer sind.

Im Gegensatz zur Zustandstabelle sind in der Transitionstabelle einige Eintragungen notwendig. Zuallererst sorgen Sie dafür, daß die beiden Transitionen `init1` *und* `init2` *als Initialelemente gekennzeichnet werden, indem Sie in den betroffenen Zeilen in der letzten Spalte* `false` *durch* `true` *ersetzen.*
Die Transition `scharfstellen` *verarbeitet die Eingabenachricht* `Scharfstellen()` *und sorgt dafür, daß die oberste LED der Statusanzeige aufleuchtet. Die Transition* `entschaerfen` *verarbeitet die gleichnamige Eingabenachricht und deaktiviert die oberste Statusleuchte. Die Transition* `ausloesen` *behandelt die Aktivierung eines Sensors (außer dem äußeren Bewegungsmelder) und reagiert darauf mit*

- *Aktivierung der Alarmsirene (*`GeraetAn(Sirene)`*)*
- *Anschalten der zweiten LED (*`LEDAn(2)`*)*
- *Anzeige des ausgelösten Sensors (*`LEDAn(stled,3+nr)`*)*

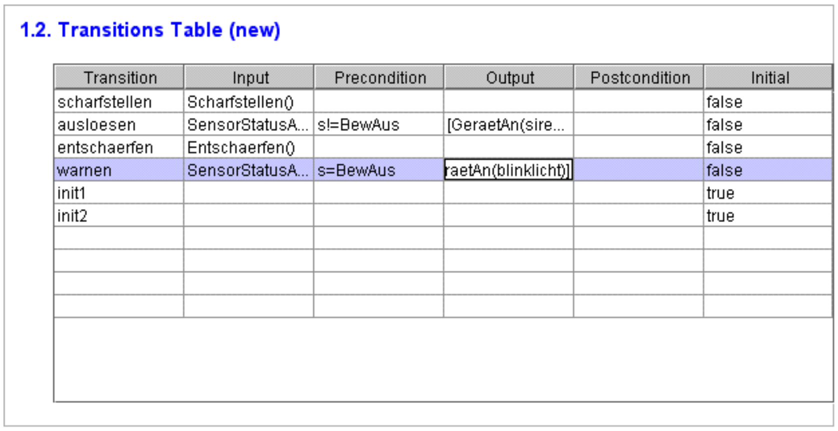
Eine Auslösung des äußeren Bewegungsmelders führt zu einem Stillen Alarm. Die komplette Transitionstabelle ist in Tabelle 3.1 zusammengefaßt.

Transition	Input	Prec.	Output	Pc.	Initial
scharfstellen	Scharfstellen()		[LedAn(stled,1)]		false
ausloesen	SensorStatus-Aenderung(s,nr,true)	s!=bewAus	[GeraetAn(sirene), LedAn(stled,2),LedAn(stled,3+nr)]		false
entschaerfen	Entschaerfen()		[LedAus(stled,1)]		false
warnen	SensorStatus-Aenderung(s,nr,true)	s=bewAus	[GeraetAn(blinklicht)]		false
init1					true
init2					true

Tabelle 3.1.: Tutorial: Transitionstabelle

3.2.4. Bearbeiten der Deklarationen

Um für das zu bearbeitende Automatendokument einen vollständigen Kontext herzustellen, sind einige Informationen notwendig, deren Bearbeitung nicht in die Verantwortung eines Editors für Automatendokumente fällt. Da die dafür notwendigen Werkzeuge derzeit nicht verfügbar sind, wird das Automatendokument übergangsweise um einen Textbereich erweitert, in dem die zusätzlich benötigten Informationen manuell nach einer sehr einfachen Syntax eingetragen werden können.
Der Inhalt des Textfeldes gliedert sich in vier Bereiche auf. Der Beginn eines jeden Bereiches ist durch eine Überschrift von den anderen getrennt. Jeder Bereich enthält eine Liste von Deklarationen eines bestimmten Typs. Die genaue



1.2. Transitions Table (new)

Transition	Input	Precondition	Output	Postcondition	Initial
scharfstellen	Scharfstellen()				false
ausloesen	SensorStatusA...	s!=BewAus	[GeraetAn(sire...		false
entschaerfen	Entschaerfen()				false
warnen	SensorStatusA...	s=BewAus	raetAn(blinklicht)]		false
init1					true
init2					true

Abbildung 3.13.: Die Zustandstabelle

Syntax der Bereiche ist abhängig von der für das Automatendokument gültigen Sprachunterstützung, die Bereichsüberschriften dagegen sind fest.
Der erste Bereich enthält eine Aufzählung aller bekannten Agententypen, im konkreten[2] Fall eine simple Aufzählung aller verwendeten Klassen, die wegen fehlender Unterstützung objektorientierter Sprachkonzepte intern auf den Typ `Int` abgebildet werden.
Der zweite Bereich enthält die Typdeklarationen aller bekannten Attribute, dritter und vierter Bereich die Signaturen für Ein- und Ausgabemethoden.

Im Deklarationsfeld ergänzen Sie nun alle Informationen, die notwendig sind, um das momentane Automatendokument zu vervollständigen.
Dazu gehört eine Liste aller referenzierten Klassen, aller Attribute, Ein- und Ausgabemethoden (siehe Tabelle 3.2). Das momentane Ergebnis könnte man jetzt gut verwenden, um mit dem Auftraggeber über das generelle Verhalten der Komponente zu diskutieren.

3.3. Verfeinerung eines Automaten

Verfeinerung eines Automaten bedeutet vor allem, daß sich das Verhalten des Automaten nicht mehr abändern darf. Verändert wird einzig und allein der Grad der Unterspezifikation, d.h. es ist möglich, Verhalten für eine bisher nicht betrachtete Kombination aus Zustand und Eingabe hinzuzufügen, oder bestehendes Verhalten zu spezialisieren, aber nicht, bestehendes Verhalten zu entfernen.
Um dies zu erreichen, werden die Möglichkeiten, das Automatendiagramm zu

[2] exemplarische Implementierung der Sprachunterstützung für FriscoF

```
TYPES:
Sensor
Geraet
LED
ATTRIBUTES:
fenster1::Sensor
fenster2::Sensor
tuere::Sensor
bewIn::Sensor
bewAus::Sensor
stled::LED
sirene::Geraet
blinklicht::Geraet
IN:
Scharfstellen()
Entschaerfen()
Loeschen()
SensorStatusAenderung(s::Sensor,nr::Int,st::Bool)
SensorFehler(s::Sensor,nr::Int)
OUT:
LedAn(l::LED,nr::Int)
LedAus(l::LED,nr::Int)
LedClear(l::LED)
GeraetAn(g::Geraet)
GeraetAus(g::Geraet)
```

Tabelle 3.2.: Tutorial: Deklarationen der Klasse `alarm`

editieren, stark beschränkt. Es stehen acht Regeln zur Verfügung, wovon jede eine festgelegte Abfolge von Änderungsoperationen und deren Voraussetzungen definiert. Diese Voraussetzungen werden entweder automatisch überprüft, oder, falls dies nicht einfach möglich ist, in Form einer Beweisverpflichtung ausgegeben, die dann beispielsweise von einem Beweiser überprüft werden muß.
Zu den Voraussetzungen gehört, daß der Automat vor Beginn einer Regelanwendung einen korrekten Kontext besitzt, die Voraussetzung der jeweiligen Regel erfüllt und nach Abschluß der Kontext wiederum korrekt ist, damit weitere Regeln angewendet werden können.
Das bedeutet, daß vor Beginn der Verfeinerung zunächst ein korrekter Kontext bewiesen werden muß. Dies geschieht durch die automatische Überprüfung von Syntax und Kontext und einmalige Ausgabe von Beweisverpflichtungen für nicht mit Bordmitteln überprüfbare Kontextbedingungen.
Nach jeder Regelanwendungen werden weitere Prüfungen geringeren Umfangs vorgenommen, die sichern, daß die zu Beginn bewiesenen Kontextbedingungen auch nach Anwendung der Regel ihre Gültigkeit behalten.
Weitere Regeln dürfen erst dann angewendet werden, wenn die Prüfungen mit positivem Ergebnis abgeschlossen worden sind, d.h. die Beweisverpflichtungen als richtig bewiesen wurden.

Sie und Ihr Auftraggeber sind mit dem momentanen Grundkonzept zufrieden und beschließen, jetzt das Automatendokument zu fixieren, d.h. Attribute, Schnittstelle und Verhalten dürfen sich nicht mehr ändern. Auf den ersten Blick könnte es vielleicht scheinen, daß man besser beraten wäre, die Festlegung auf den Verfeinerungsmodus auf einen späteren Zeitpunkt zu verschieben, um sich möglichst viel Flexibilität zu erhalten. Andererseits möchte man ja bestimmte Eigenschaften des Automaten bewiesen haben und das könnte sich bei einem hohen Detaillierungsgrad als sehr schwierig erweisen. Im momentanen einfachen Grundzustand fällt der Beweis sicher viel einfacher aus und der Verfeinerungsmodus ist so gestaltet, daß bei jedem Verfeinerungsschritt nur ein auf diesen Schritt bezogener Beweis notwendig wird. Aus Erfahrung wissen Sie, daß es viel leichter ist, mehrere einfache Aussagen zu Beweisen, als eine komplexe, besonders wenn man auf automatische Beweiser zurückgreifen will. Außerdem kann man nie wissen, ob sich in einem späteren Stadium der unüberprüften Erweiterung eines dann komplexeren Automatenmodells nicht doch unabsichtlich eine unerwünschte Verhaltensänderung einschleicht.
In jedem Fall beschließen Sie, das Dokument zunächst durch Aufruf des Menüpunktes `Document|Save` *zu speichern, damit Sie, falls später doch noch ein weiteres Attribut oder eine grundlegende Änderung notwendig werden sollte, an diese Stelle zurückkehren können. Sämtliche Verfeinerungsschritte werden ja als Skript gespeichert und können jederzeit automatisch bis an eine Stelle wiederholt werden, wo sich durch die nachträgliche Abänderung ein Fehler ergibt.*
Um in den Verfeinerungsmodus zu wechseln, klicken Sie im Kontrollpart am Ende

des Dokumentes auf das Kästchen neben `refine`*. Das Dokument wird zuerst auf syntaktische bzw. Kontextfehler überprüft. Falls keine Fehler auftreten, befinden Sie sich im Verfeinerungsmodus (siehe Abb. 3.14).*

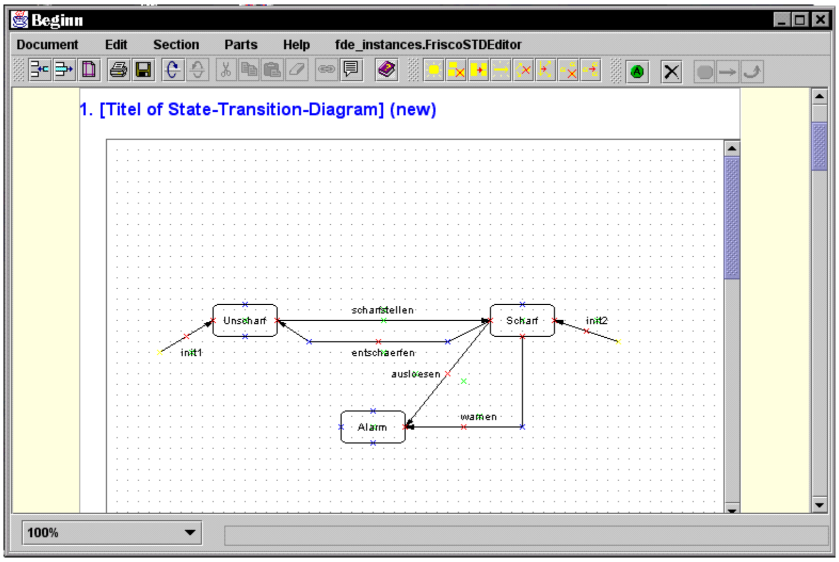


Abbildung 3.14.: Verfeinerungsmodus

3.3.1. Beweisverpflichtungen und der Beweismanager

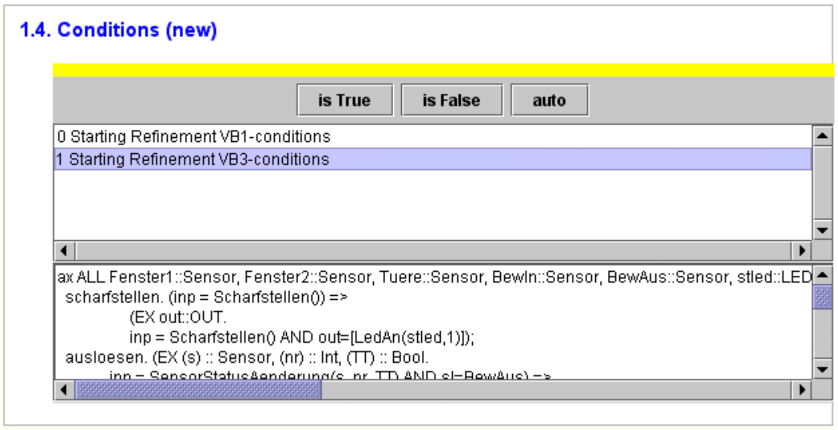


Abbildung 3.15.: Der Proof-Manager

Alle im Laufe eines Verfeinerungsprozesses auflaufenden Beweisverpflichtungen werden im Part `Proof-Manager`(Abb. 3.15) gesammelt und verwaltet.

Dort können alle Beweisverpflichtungen manuell als wahr oder falsch markiert werden, oder, falls im Idealfall ein angeschlossener Beweiser zur Verfügung steht, von diesem automatisch überprüft werden.
Im oberen Fenster wird ein Verzeichnis aller gespeicherten Beweisverpflichtungen angezeigt. Der Name des jeweiligen Verzeichniseintrages gibt einen Hinweis auf die Regelanwendung, aus der die Beweisverpflichtungen entstanden sind.
Das untere Fenster zeigt die Beweisverpflichtungen des gerade gewählten Verzeichniseintrages vollständig an. Mit den Knöpfen `isTrue` und `isFalse` kann eine Beweisverpflichtung manuell als richtig bzw. falsch markiert werden. Ein Klick auf den Knopf `auto` übergibt die momentan selektierte Beweisverpflichtung an einen angeschlossenen Beweiser[3].
Am oberen Ende des Parts befindet sich eine Statusanzeige. Dies ist:

- grau, wenn keine Beweisverpflichtungen gespeichert sind.
- grün, wenn alle Beweisverpflichtungen positiv erfüllt werden konnten.
- gelb, wenn ein oder mehrere Beweisverpflichtungen noch nicht bewiesen wurden.
- rot blinkend, wenn ein oder mehrere Beweisverpflichtungen als falsch bewiesen wurden.

Nach dem Start der Verfeinerung wurden die zwei Beweisverpflichtungen aus Tabelle 3.3 ausgegeben.
Die erste Beweisverpflichtung sorgt dafür, daß einem Zustand keine unerfüllbare Unterteilung des Datenzustandsraumes zugewiesen wurde. Da alle drei Zustände nicht näher durch ein Pattern oder ein Prädikat beschrieben sind, sind die Beweisverpflichtungen trivial.
Gleiches gilt für die zweite Beweisverpflichtung, welche die Schaltbarkeit der Transitionen überprüft. Jede der Implikationen läßt sich notfalls durch die Aufzählung der endlichen Möglichkeiten beweisen.
Bevor Sie mit dem ersten Verfeinerungsschritt beginnen, müssen zuvor die beiden Beweisverpflichtungen überprüft und als `true` *markiert werden. Dies geschieht durch das Drücken des Knopfes* `isTrue` *woraufhin sich die selektierte Beweisverpflichtung grün färbt.*

3.3.2. Hinzufügen von Zuständen

Über die Regel **addS**(Hinzufügen von Zuständen) kann eine weitere Unterteilung des Datenzustandsraumes in Äquivalenzklassen vorgenommen werden, d.h. eine

[3] Version 1.0 enthält noch keinen Beweiseranschluß

```
ax {
  Unscharf. true;
  Scharf. true;
  Alarm. true;
};
ax ALL fenster1::Sensor, fenster2::Sensor, tuere::Sensor,
       bewIn::Sensor, bewAus::Sensor, stled::LED, sirene::Geraet,
       blinklicht::Geraet, inp::IN .{
  scharfstellen. (inp = Scharfstellen()) =>
                  (EX out::[OUT].
                  inp = Scharfstellen() AND out=[LedAn(stled,1)]);
  ausloesen. (EX (s) :: Sensor, (nr) :: Int, (true) :: Bool.
              inp = SensorStatusAenderung(s, nr, true) AND s!=bewAus) =>
              (EX (s) :: Sensor, (nr) :: Int, (true) :: Bool, out::[OUT].
              inp = SensorStatusAenderung(s, nr, true) AND s!=bewAus AND
              out=[GeraetAn(sirene),LedAn(stled,2),LedAn(stled,3+nr)]);
  entschaerfen. (inp = Entschaerfen()) =>
                 (EX out::[OUT].
                 inp = Entschaerfen() AND out=[LedAus(stled,1)]);
  warnen. (EX (s) :: Sensor, (nr) :: Int, (true) :: Bool.
           inp = SensorStatusAenderung(s, nr, true) AND s=bewAus) =>
           (EX (s) :: Sensor, (nr) :: Int, (true) :: Bool, out::[OUT].
           inp = SensorStatusAenderung(s, nr, true) AND s=bewAus AND
           out=[GeraetAn(blinklicht)]);
};
```

Tabelle 3.3.: Tutorial: Beweisverpflichtungen (Beginn der Verfeinerung)

bisher nicht verwendete Kombination aus Musterangaben für bestimmte Attribute kann explizit unter einem Zustandsnamen zusammengefaßt werden.

Es ist zu beachten, daß mit dem neuen Zustand automatisch ein dem Zustandsnamen entsprechendes Kontrollattribut ausgezeichnet wird. Dieses ist nicht im Feld für das Zustandspattern in der Zustandstabelle angegeben aber nichtsdestotrotz explizit vorhanden.

Damit wird sichergestellt, daß immer eine bisher nicht verwendete Kombination des Datenzustandsraumes ausgezeichnet und damit die Disjunktheit der Aufteilung des Datenzustandsraumes gesichert bleibt.

Auf diesem Wege ist es auch möglich, zu Strukturierungszwecken reine Kontrollzustände einzuführen, ohne nachträglich neue Attribute einzufügen, was während der Verfeinerung nicht möglich ist.

Um Zustände hinzuzufügen aktivieren Sie die Regel **addS** über die Toolbar. Der betroffene Knopf der Toolbar verwandelt sich in ein Stoppschild, Knöpfe für alle anderen Regeln werden deaktiviert, ein Knopf zum Hinzufügen eines neuen Zustandes wird aktiviert, d.h. wechselt von grau nach farbig und läßt sich anwählen.

Jetzt können beliebig viele Zustände hinzugefügt werden. Um den momentanen Verfeinerungsschritt abzuschließen, drücken Sie in der Toolbar auf das Stoppschild. Die neu eingegebenen Zustände werden auf Syntax und Kontext geprüft und falls keine Fehler auftreten, werden die notwendigen Beweisverpflichtungen ausgegeben, die Regel **addS** wird beendet, die Editierfunktionen deaktiviert und die Regelauswahl wieder aktiviert. Traten Fehler auf, werden diese im Kontrollpart ausgegeben, was dieser mit einer rot blinkenden Fehleranzeige quittiert. Sobald sie verbessert sind, kann durch Anwahl der Stoppfunktion erneut versucht werden, die Regel und damit den Verfeinerungsschritt zu beenden. Ein neuer Verfeinerungsschritt kann erst begonnen werden, wenn die Beweisverpflichtungen alle als `true` markiert wurden.

Jetzt beginnen Sie mit dem ersten Verfeinerungsschritt. Bis jetzt war die Frage der Fehlerbehandlung ungeklärt, also die Frage, was passieren soll, falls ein Sensor einen Fehler meldet.

Sie entscheiden sich dafür, bei Meldung eines Fehlers in einen Fehlerzustand zu wechseln, damit durch einen defekten Sensor kein Fehlalarm ausgelöst wird.

Diesen Fehlerzustand führen Sie nun ein, indem Sie die Funktion `addS` *entweder über das Menü oder die Werkzeugleiste aktivieren. Nun können Sie wie schon beim Zeichnen des Grundmodells (Abschnitt 3.2.1) einen neuen Zustand hinzufügen, den Sie mit* `Stoerung` *bezeichnen. Der Zustand ist wiederum ein reiner Kontrollzustand, so daß keine weiteren Einträge in die Zustandstabelle notwendig sind.*

Da Sie im Moment keine weiteren Zustände hinzufügen wollen, beenden Sie diesen Verfeinerungsschritt, indem Sie die Funktion `stop refS` *wählen. Es wird eine neue Beweisverpflichtung ausgegeben, die aufgrund der Tatsache, daß es sich bei dem neuen Zustand um einen Kontrollzustand handelt, trivial ist.*

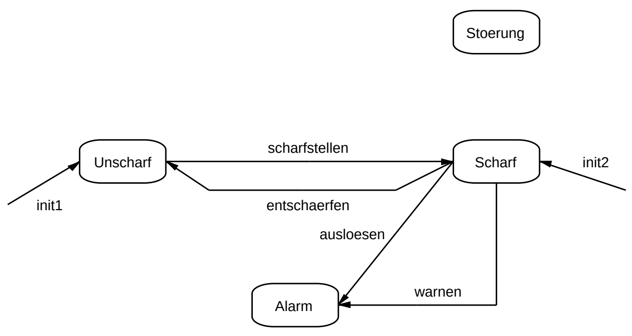


Abbildung 3.16.: Tutorial: Diagramm nach erstem Schritt (addS)

3.3.3. Löschen von Zuständen

Das Löschen von Zuständen (Anwendung der Regel **remS**) funktioniert analog zur Regel remS (Abschnitt 3.3.2), mit dem Unterschied, daß an Editierfunktionen nur das Löschen zur Verfügung steht. Der wichtigste Unterschied ist, daß im Gegensatz zum Hinzufügen von Zuständen jeder Aufruf der Löschfunktion einen kompletten Verfeinerungsschritt markiert. Aufgrund der Beschaffenheit der Löschregel kann der zugehörige Verfeinerungsschritt nicht iterativ aufgebaut werden. Statt dessen ist es möglich, mehrere Zustände zu selektieren und diese auf einmal zu löschen. Sollte versucht werden, mit der Regel **remS** Transitionen oder Initialelemente zu löschen, die als Zielzustand einen nicht zu löschenden Zustand haben, wird eine entsprechende Fehlermeldung ausgegeben. Das Löschen von Transitionen zusammen mit ihren Quellzuständen ist aber möglich, da diese im Anschluß sowieso automatisch entfernt würden.

3.3.4. Hinzufügen und Löschen von Transitionen

Das Hinzufügen respektive Löschen von Transitionen ist völlig analog zu den Regeln addS bzw. remS, nur daß statt Zuständen Transitionen hinzugefügt respektive gelöscht werden, die Editiermöglichkeiten also entsprechend anders freigegeben werden.

Nachdem nun ein Kontrollzustand für die Behandlung von Fehlermeldungen vorhanden ist, muß jetzt natürlich dafür gesorgt werden, daß die entsprechenden Eingabenachrichten verarbeitet werden.
Sie entscheiden, daß eine Auswertung der Fehlermeldungen vorerst nur interessant ist, wenn sich die Alarmanlage im Zustand `Scharf` *befindet. Beginnen Sie also einen Verfeinerungsschritt nach der Regel* `addT`*. Der Rest funktioniert analog zum letzten Schritt, nur daß Sie diesmal eine Transition hinzufügen. Da die Kontextprüfung aktiv ist, wird mit Recht moniert, daß die eben hinzugefügte Transition noch kein Eingabepattern besitzt. Sie wird deshalb rot eingefärbt und mit einer Anmerkung versehen (Abb. 3.17). Wenn Sie den Mauszeiger über die Annotation (das eingekreiste große A) bewegen, können Sie in der Statuszeile die Fehlermeldung lesen, ohne erst in der Fehlerliste am unteren Ende des Dokumentes nachsehen zu müssen.*
Tragen Sie also in der Transitionstabelle die benötigten Daten, in diesem Fall Ein- und Ausgabe ein. Im Fehlerfall soll neben der gelben LED, welche eine Störung anzeigt auch die LED des jeweiligen Sensors aufleuchten, der die Störung verursacht hat.
Weil Sie gerade beim Hinzufügen von Transitionen sind, entschließen Sie sich, auch gleich die Möglichkeit einzubauen, die von der Funkbedienung ausgelöste Nachricht `Loeschen()` *dazu zu benutzen, von den Ausnahmezuständen* `Alarm` *und* `Stoerung` *in den nächst logischen Grundzustand zurückzufallen. Sie fügen*

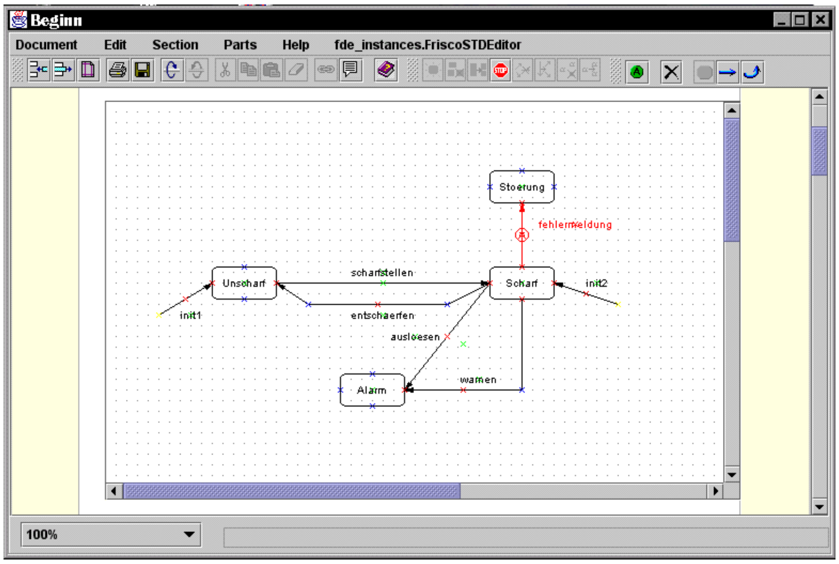


Abbildung 3.17.: Tutorial: Hinzufügen einer Transition (addT)

eine weitere Transition `loeschen1` *hinzu, die im Falle einer Störung wieder in den unscharfen Zustand wechselt und alle LEDs der Statusanzeige löscht. Die Transition* `loeschen2` *erledigt ähnliches im Alarmfall, nur daß sie in den scharfen Zustand zurückfällt und zusätzlich die Sirene abschaltet. Da im Zustand Alarm unterschiedliche LEDs brennen können, ist es einfacher, zunächst alle Statusleuchten zu löschen und dann diejenige, welche den scharfen Zustand anzeigt, wieder zu aktivieren, als die während des Alarms aktivierten LEDs einzeln abzuschalten (siehe Tabelle 3.4).*
Nach dem Beenden des Verfeinerungsschrittes (`stop addT`*) wird eine Beweisverpflichtung ausgegeben, die dafür sorgt, daß einerseits die neuen Transitionen schaltbar sind, andererseits kein weiterer Nichtdeterminismus dergestalt eingeführt wird, daß für einen Zustand und eine Eingabenachricht plötzlich eine weitere Schaltmöglichkeit besteht (siehe Tabelle 3.5).*
Für die Transitionen `loeschenX` *muß natürlich kein Nichtdeterminismus überprüft werden, da sie die jeweils einzigen Transitionen mit ihrem Quellzustand sind.*

3.3.5. Verfeinern von Zuständen

Das Verfeinern von Zuständen (Regel **refS**) ist prinzipiell identisch mit dem Verfahren, Zustände hinzuzufügen. Einziger Unterschied ist, daß vor Aufruf dieser Regel genau ein Zustand markiert sein muß, der durch die neu hinzugefügten Zustände ersetzt, d.h. verfeinert werden soll.

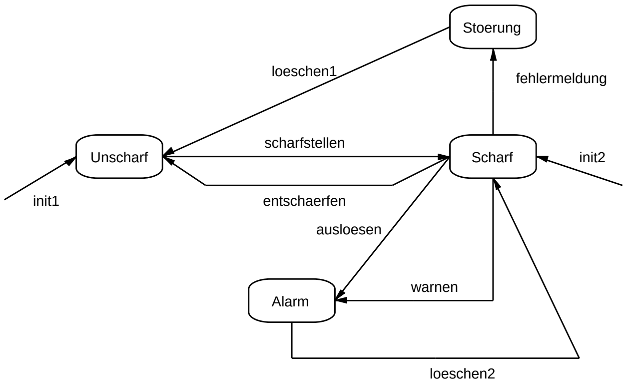


Abbildung 3.18.: Tutorial: Diagramm nach zweitem Schritt (addT)

Transition	Input	Precondition	Output	Pc.	Ini
scharfstellen	Scharfstellen()		[LedAn(stled,1)]		false
ausloesen	SensorStatus-Aenderung(s,nr,true)	s!=bewAus	[GeraetAn(sirene), LedAn(stled,2),LedAn(stled,3+nr)]		false
entschaerfen	Entschaerfen()		[LedAus(stled,1)]		false
warnen	SensorStatus-Aenderung(s,nr,true)	s=bewAus	[GeraetAn(blinklicht)]		false
init1					true
init2					true
fehlermeldung	SensorFehler(s,nr)		[LedAn(stled,3), LedAn(stled,3+nr)]		false
loeschen1	Loeschen()		[LedClear(stled)]		false
loeschen2	Loeschen()		[LedClear(stled), LedAn(stled,1)]		false

Tabelle 3.4.: Tutorial: Transitionstabelle (2. Schritt - addT)

```
ax ALL fenster1::Sensor, fenster2::Sensor, tuere::Sensor,
       bewIn::Sensor, bewAus::Sensor, stled::LED, sirene::Geraet,
       blinklicht::Geraet, inp::IN . {
  fehlermeldung. (EX (s) :: Sensor, (nr) :: Int.
                  inp = SensorFehler(s, nr)) =>
                  (EX (s) :: Sensor, (nr) :: Int, out::[OUT].
                  inp = SensorFehler(s, nr) AND out=[LedAn(stled,3),
                  LedAn(stled,3+nr)]);
  loeschen1. (inp = Loeschen()) =>
              (EX out::[OUT].
              inp = Loeschen() AND out=[LedClear(stled)]);
  loeschen2. (inp = Loeschen()) =>
              (EX out::[OUT].
              inp = Loeschen() AND out=[LedClear(stled),LedAn(stled,1)]);
  ausloesen.  NOT (EX (s) :: Sensor, (nr) :: Int.
               inp = SensorFehler(s, nr)) AND (EX (s) :: Sensor, (nr) :: Int,
               (true) :: Bool.
               inp = SensorStatusAenderung(s, nr, true) AND s!=bewAus));
  entschaerfen.  NOT (EX (s) :: Sensor, (nr) :: Int.
                  inp = SensorFehler(s, nr)) AND (inp = Entschaerfen()));
  warnen.  NOT (EX (s) :: Sensor, (nr) :: Int.
            inp = SensorFehler(s, nr)) AND (EX (s) :: Sensor, (nr) :: Int,
             (true) :: Bool.
            inp = SensorStatusAenderung(s, nr, true) AND s=bewAus));
};
```

Tabelle 3.5.: Tutorial: Beweisverpflichtungen (2. Schritt - addT)

Nach Beendigung des Verfeinerungsschrittes wird der Eingangs markierte Zustand gelöscht und alle mit ihm verbundenen Transitionen und Initialelemente entsprechend vervielfacht und mit den neuen Zuständen verbunden.

Als nächstes haben Sie sich die genauere Modellierung des Annäherungsalarms vorgenommen. Bisher waren der stille und laute Alarm in einem Zustand vereint, es war aber nicht geklärt, ob während eines stillen Alarms auch ein lauter Alarm ausgelöst werden kann. Dies könnte man durch Hinzufügen einer Transition erledigen, die vom Zustand Alarm in einer Schleife zu diesem zurückführt.

```
ax {
  AlarmStill. true;
  AlarmLaut. true;
  Alarm. (true) <=> ((true) OR (true));
};
```

Tabelle 3.6.: Tutorial: Beweisverpflichtungen (3. Schritt - refS)

Für eine bessere Übersichtlichkeit entschließen Sie sich aber, den Zustand Alarm aufzuteilen. Dazu selektieren Sie zunächst den Zustand `Alarm` und aktivieren dann die Verfeinerungsregel `refS`. Jetzt fügen Sie die zwei Zustände `AlarmStill`

Transition	Input	Precondition	Output	PC.	Ini
scharfstellen	Scharfstellen()		[LedAn(stled,1)]		f
entschaerfen	Entschaerfen()		[LedAus(stled,1)]		f
init1					t
init2					t
fehlerm.	SensorFehler(s,nr)		[LedAn(stled,3), LedAn(stled,3+nr)]		f
loeschen1	Loeschen()		[LedClear(stled)]		f
ausloesen_0	SensorStatus-Aenderung(s,nr,true)	s!=bewAus AND (EX (s) :: Sensor, (nr) :: Int, (true) :: Bool, out::[OUT]. inp = SensorStatusAenderung(s, nr, true) AND s!=bewAus AND out=[GeraetAn(sirene), LedAn(stled,2),LedAn(stled,3+nr)])	[GeraetAn(sirene),] LedAn(stled,2), LedAn(stled,3+nr)		f
ausloesen_1	SensorStatus-Aenderung(s,nr,true)	s!=bewAus AND (EX (s) :: Sensor, (nr) :: Int, (true) :: Bool, out::[OUT]. inp = SensorStatusAenderung(s, nr, true) AND s!=bewAus AND out=[GeraetAn(sirene), LedAn(stled,2),LedAn(stled,3+nr)])	[GeraetAn(sirene), LedAn(stled,2), LedAn(stled,3+nr)]		f
warnen_0	SensorStatus-Aenderung(s,nr,true)	s=bewAus AND (EX (s) :: Sensor, (nr) :: Int, (true) :: Bool, out::[OUT]. inp = SensorStatusAenderung(s, nr, true) AND s=bewAus AND out=[GeraetAn(blinklicht)])	[GeraetAn(blinklicht)]		f
warnen_1	SensorStatus-Aenderung(s,nr,true)	s=bewAus AND (EX (s) :: Sensor, (nr) :: Int, (true) :: Bool, out::[OUT]. inp = SensorStatusAenderung(s, nr, true) AND s=bewAus AND out=[GeraetAn(blinklicht)])	[GeraetAn(blinklicht)]		f
loeschen2_0	Loeschen()	(EX out::[OUT]. inp = Loeschen() AND out=[LedClear(stled),LedAn(stled,1)])	[LedClear(stled) ,LedAn(stled,1)]		f
loeschen2_1	Loeschen()	(EX out::[OUT]. inp = Loeschen() AND out=[LedClear(stled),LedAn(stled,1)])	[LedClear(stled) ,LedAn(stled,1)]		f

Tabelle 3.7.: Tutorial: Transitionstabelle (3. Schritt - refS)

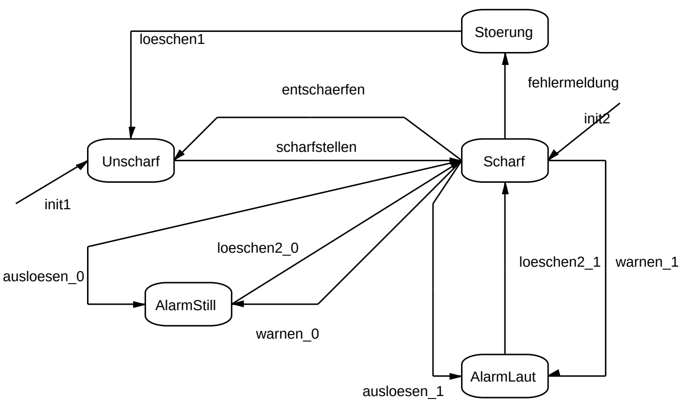


Abbildung 3.19.: Tutorial: Diagramm (3. Schritt - refS)

und `AlarmLaut` *hinzu. Den alten Zustand* `Alarm` *brauchen Sie nicht zu löschen. Beenden Sie einfach den Verfeinerungsschritt und das Programm löscht automatisch den alten Zustand, verdoppelt all seine Transitionen und verbindet Sie mit den neuen Zuständen. Jetzt müssen Sie nur noch die jeweils unerwünschten Transitionen löschen, damit auch wirklich eine neue Strukturierung entsteht. Aktivieren Sie also die Regel* `remT` *und löschen Sie die Transitionen* `warnen_1` *und* `ausloesen_0` *Dies ist ohne weiteres möglich, da ja durch das vorausgegangene Klonen für jede der gelöschten Transition ein Zwilling bereitsteht, der eine Kombination aus Eingabe und Quellzustand abdeckt (Tabelle 3.8). Der Beweis gestaltet sich, auch wenn die Beweisverpflichtungen umfangreich erscheinen mögen, entsprechend einfach, da sich die Vorbedingung der gelöschten Transition jeweils 1:1 in der Veroderung aller übrigen Vorbedingung wiederfinden.*

```
ax ALL fenster1::Sensor, fenster2::Sensor, tuere::Sensor,
       bewIn::Sensor, bewAus::Sensor, stled::LED, sirene::Geraet,
       blinklicht::Geraet, inp::IN .{
  warnen_1. (EX (s) :: Sensor, (nr) :: Int, (true) :: Bool.
             inp = SensorStatusAenderung(s, nr, true) AND
             s=bewAus AND (EX (s) :: Sensor, (nr) :: Int,
             (true) :: Bool, out::[OUT].
             inp = SensorStatusAenderung(s, nr, true) AND
             s=bewAus AND out=[GeraetAn(blinklicht)])) =>

             ((inp = Entschaerfen()) OR

             (EX (s) :: Sensor, (nr) :: Int.
             inp = SensorFehler(s, nr)) OR

             (EX (s) :: Sensor, (nr) :: Int, (true) :: Bool.
             inp = SensorStatusAenderung(s, nr, true) AND
             s!=bewAus AND (EX (s) :: Sensor, (nr) :: Int,
              (true) :: Bool, out::[OUT].
             inp = SensorStatusAenderung(s, nr, true) AND s!=bewAus AND
             out=[GeraetAn(sirene),LedAn(stled,2),LedAn(stled,3+nr)])) OR

             (EX (s) :: Sensor, (nr) :: Int, (true) :: Bool.
             inp = SensorStatusAenderung(s, nr, true) AND
             s!=bewAus AND(EX (s) :: Sensor, (nr) :: Int,
              (true) :: Bool, out::[OUT].
             inp = SensorStatusAenderung(s, nr, true) AND s!=bewAus AND
             out=[GeraetAn(sirene),LedAn(stled,2),LedAn(stled,3+nr)])) OR

             (EX (s) :: Sensor, (nr) :: Int, (true) :: Bool.
             inp = SensorStatusAenderung(s, nr, true) AND
             s=bewAus AND(EX (s) :: Sensor, (nr) :: Int,
              (true) :: Bool, out::[OUT].
             inp = SensorStatusAenderung(s, nr, true) AND
             s=bewAus AND out=[GeraetAn(blinklicht)])));
};
```

Tabelle 3.8.: Tutorial: Beweisverpflichtungen (4. Schritt - remT)

```
ax ALL fenster1::Sensor, fenster2::Sensor, tuere::Sensor,
       bewIn::Sensor, bewAus::Sensor, stled::LED, sirene::Geraet,
       blinklicht::Geraet, inp::IN .{
  ausloesen_0. (EX (s) :: Sensor, (nr) :: Int, (true) :: Bool .
                inp = SensorStatusAenderung(s, nr, true) AND
                s!=bewAus AND(EX (s) :: Sensor, (nr) :: Int,
                (true) :: Bool, out::[OUT].
                inp = SensorStatusAenderung(s, nr, true) AND s!=bewAus AND
                out=[GeraetAn(sirene),LedAn(stled,2),LedAn(stled,3+nr)])) =>

                ((inp = Entschaerfen()) OR

                (EX (s) :: Sensor, (nr) :: Int.
                inp = SensorFehler(s, nr)) OR

                (EX (s) :: Sensor, (nr) :: Int, (true) :: Bool.
                inp = SensorStatusAenderung(s, nr, true) AND
                s!=bewAus AND(EX (s) :: Sensor, (nr) :: Int,
                (true) :: Bool, out::[OUT].
                inp = SensorStatusAenderung(s, nr, true) AND s!=bewAus AND
                out=[GeraetAn(sirene),LedAn(stled,2),LedAn(stled,3+nr)])) OR

                (EX (s) :: Sensor, (nr) :: Int, (true) :: Bool.
                inp = SensorStatusAenderung(s, nr, true) AND
                s=bewAus AND(EX (s) :: Sensor, (nr) :: Int,
                (true) :: Bool, out::[OUT].
                inp = SensorStatusAenderung(s, nr, true) AND
                s=bewAus AND out=[GeraetAn(blinklicht)])));
};
```

Tabelle 3.9.: Tutorial: Beweisverpflichtungen (5. Schritt - remT)

Transition	Input	Precondition	Output	Pc	Ini
scharfstellen	Scharfstellen()		[LedAn(stled,1)]		f
entschaerfen	Entschaerfen()		[LedAus(stled,1)]		f
init1					t
init2					t
fehlermeldung	SensorFehler(s,nr)		[LedAn(stled,3), LedAn(stled,3+nr)]		f
loeschen1	Loeschen()		[LedClear(stled)]		f
ausloesen_1	SensorStatus- Aenderung(s,nr,true)	s!=bewAus AND (EX (s) :: Sensor, (nr) :: Int, (true) :: Bool, out::[OUT]. inp=SensorStatusAenderung(s, nr, true) s!=bewAus AND out=[GeraetAn(sirene), LedAn(stled,2),LedAn(stled,3+nr)])	[GeraetAn(sirene), LedAn(stled,2), LedAn(stled,3+nr)]		f
warnen_0	SensorStatus- Aenderung(s,nr,true)	s=bewAus AND (EX (s) :: Sensor, (nr) :: Int, (true) :: Bool, out::[OUT]. inp=SensorStatusAenderung(s, nr, true) s=bewAus AND out=[GeraetAn(blinklicht)])	[GeraetAn(blinklicht)]		f
loeschen2_0	Loeschen()	(EX out::[OUT]. inp = Loeschen() AND out=[LedClear(stled),LedAn(stled,1)])	[LedClear(stled), LedAn(stled,1)]		f
loeschen2_1	Loeschen()	(EX out::[OUT]. inp = Loeschen() AND out=[LedClear(stled),LedAn(stled,1)])	[LedClear(stled), LedAn(stled,1)]		f

Tabelle 3.10.: Tutorial: Transitionstabelle (5. Schritt - remT)

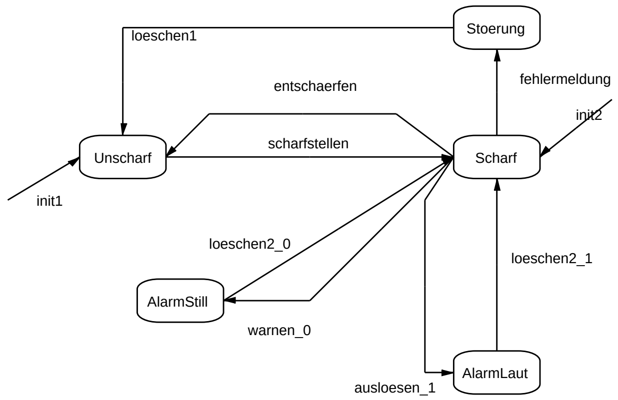


Abbildung 3.20.: Tutorial: Diagramm (5. Schritt - addT)

3.3.6. Verfeinern von Transitionen

Das Verfeinern von Transitionen geschieht analog zum Hinzufügen, nur daß andere Beweisverpflichtungen zu erfüllen sind, die sicherstellen, daß die neue Transition eine Spezialisierung bereits vorhandene Transitionen ist. Im Unterschied zum Verfeinern von Zuständen wird keine automatische Löschung vorgenommen, weshalb auch vor Aktivierung des Schritts keine Transition ausgewählt werden muß.

3.3.7. Löschen und Verfeinern von Initialelementen

Das Verfeinern respektive Löschen von Initialelementen ist völlig analog zu den Regeln refT bzw. remT, nur daß statt Transitionen Initialelemente verfeinert respektive gelöscht werden.

Sie sind mit den bisherigen Fortschritten für heute zufrieden und lehnen sich zurück, als das Telefon klingelt. Ihr Kollege aus der Hardwareabteilung gibt Antwort auf Ihre Anfrage vom Vormittag und erklärt Ihnen, daß aufgrund einer autarken Energieversorgung nicht mit einem Ausfall der Anlage zu rechnen ist.
Sie beschließen daraufhin, sich für die Initialisierung in den unscharfen Zustand zu entscheiden. Die Initialtransition `init2` *kann gelöscht werden. Wählen Sie also die Verfeinerung* **`remI`** *und entfernen Sie das zweite Initialelement. Die einzige entstehende Beweisverpflichtung, nämlich daß mindestens eine Initialtransition vorhanden sein muß, wird vom Programm automatisch überprüft. Zufrieden speichern Sie das Ergebnis und machen für heute Schluß.*

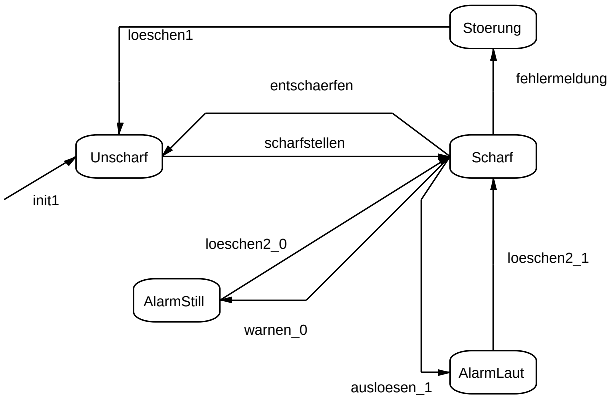


Abbildung 3.21.: Tutorial: Diagramm (6. Schritt - remI)

4. Systemanforderungen

In diesem Kapitel wird zunächst die Aufgabenstellung im Sinne einer Analyse konkretisiert. Im Anschluß daran werden die für jede Komponente gültigen Anforderungen herausgearbeitet, erläutert und die sich dann daraus ergebende abstrakte Spezifikation des Systems entwickelt. Eine Beschreibung der konkreten Umsetzung findet sich in Abschnitt 5 ab Seite 79.

4.1. Aufgabenstellung

Diese Arbeit im Rahmen des Frisco-Projektes beschreibt die Implementierung eines Automateneditors, der die konkrete Syntax einer bereits vorher im Rahmen dieses Projektes entwickelten Notation für Verhaltensautomaten realisiert. Weiterhin wurde ein ebenfalls vorhandener Kalkül für semantikerhaltende Verfeinerung dieser Automaten zusammen mit geeigneten Kontext- und Syntaxprüfungen in diesen Editor implementiert.
Als Laufzeitumgebung des Editors ist die Werkzeugplattform OEF vorgesehen, d.h. das Werkzeug muß schlußendlich als installierbarer und lauffähiger Part vorliegen.
Da Transitionen mit Vor- und Nachbedingungen in PL.1 attributiert sind, sind bei Bedarf Verifikationsbedingungen auszugeben. Zusätzlich sind Vorbereitungen für die Übergabe der Verifikationsbedingungen an einen Beweiser zu treffen.
Die Verarbeitung der Vor- und Nachbedingungen sowie der Zustandspattern des Automaten erfolgen über den Parser einer funktionalen Sprache, der geeignet anzuschließen ist. Dabei ist darauf zu achten, daß keine Festlegung auf eine konkrete funktionale Sprache erfolgt und ein Wechsel der verwendeten Sprache jederzeit durch einfachen Austausch des Parsermoduls möglich ist.

4.2. Erläuterungen zur Aufgabenstellung

Die dem zu entwickelnden Werkzeug zugrundeliegende Theorie ist nur ein Ausschnitt eines größeren Ganzen, folglich ist auch das Werkzeug auf die Basis einer offenen Architektur gestellt worden, so daß die anderen Teile später ohne Schwierigkeit ergänzt werden können. Konkret wurde die Werkzeugunterstützung für

eine der in [33] beschriebenen Dokumentarten, nämlich den Automatendokumenten, implementiert.
Die einzelnen Dokumentarten sind aber keine abgeschlossenen Einheiten. Das bedeutet, daß einige Informationen aus anderen Dokumentarten benötigt werden. Konkret handelt es sich um eine Liste von Attributen, Objektreferenzen sowie Signaturen von Ein- und Ausgabemethoden. Da noch keine Werkzeuge für die anderen Dokumentarten existieren, mußte auch ein (einfacher) Editor zur Verfügung gestellt werden, der die Eingabe dieser Informationen zuläßt.
Das als Ablaufumgebung vorgesehene OEF ist eine stark komponentenbasierte Plattform, wodurch weitere Konzepte wie Wiederverwendbarkeit und Austauschbarkeit in die Überlegungen mit einbezogen werden mußten. Für die Dokumentart 'Automatendokument' werden Editoren für Automatendiagramme, Tabellen und Text benötigt. Da alle Editoren bereits als OEF-Komponenten zur Verfügung standen, machte es wenig Sinn, neue Editoren zu entwickeln, die ganz speziell auf den Verwendungszweck zugeschnitten sind. Statt dessen war es erwünscht, auf bestehende Komponenten zurückzugreifen und diese bei Bedarf um Funktionalität zu erweitern.

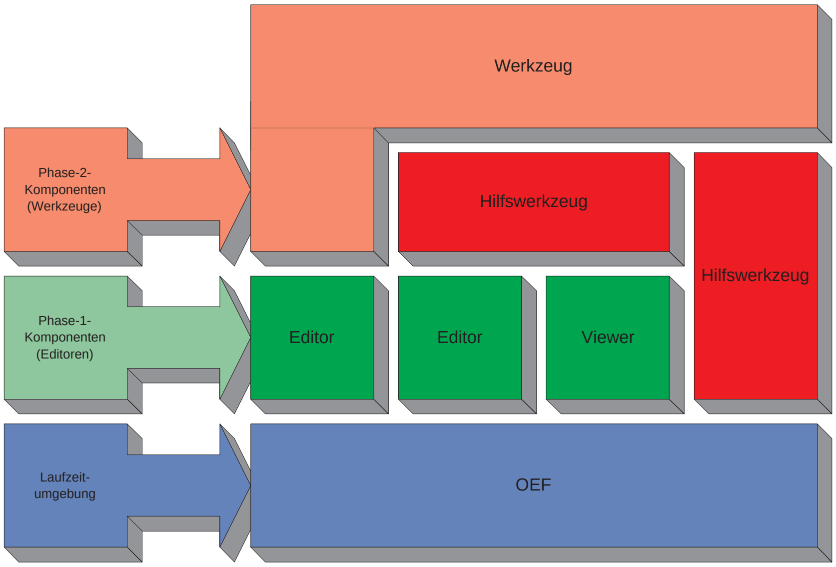


Abbildung 4.1.: Komponentenarchitektur eines OEF-Werkzeuges

Gemäß den Vorgaben zur Entwicklung von OEF-Komponenten wurde das Werkzeug in semantikfreie, bzw. semantikverarbeitende Komponenten aufgeteilt. Dies geschah durch die Verwendung von allgemein anwendbaren Editorkomponenten zur Ein- und Ausgabe von semantikfreien Informationen (Text, Tabellen, Diagramme) (Phase-1-Komponenten) und der Neuentwicklung einer spezialisierten Kontroll- und Verarbeitungskomponente, die alle benötigten Editoren kontrolliert, deren semantikfreie Information mit Semantik anreichert und verarbeitet

(Phase-2-Komponenten). Die daraus folgende allgemein anwendbare konzeptionelle Komponentenarchitektur für OEF-Werkzeuge zeigt Abbildung 4.1.
Damit wird auch für zukünftige Werkzeuge eine optimale Wiederverwendbarkeit (es existiert in der Regel jeweils nur ein Editor für eine bestimmte Informationsart) sowie optimale Austauschbarkeit (ein bestehender Editor kann jederzeit durch eine modernere Variante ersetzt werden) erreicht. Neue Funktionen sind sofort ohne Änderung in allen Werkzeugen verfügbar. Außerdem existiert eine einheitliche Benutzerführung der Editoren in jedem Werkzeug.
Art und Umfang der Aufgabenstellung erforderten die Kooperation mit weiteren eigenständigen Phase-2 Werkzeugkomponenten. Konkret handelt es sich um einen als Hilfswerkzeug implementierten Parser für eine funktionale Sprache sowie ein Werkzeug zur Verwaltung von Verifikationsbedingungen und deren Beweise (später soll einmal ein Beweiser hinzukommen).
Die so vorgegebene Aufteilung in austauschbare Einzelkomponenten, die ein Zwei-Schichten-Modell für die Informationsverarbeitung induziert, bedingte die Entwicklung eines erweiterten Model-View-Mechanismus.
Neben der eigentlichen Arbeit, dem Entwurf einer offenen Architektur und der Implementierung des Kernsystems mußten alle peripheren Komponenten, die noch nicht entwickelt worden waren, zumindest so weit durch Platzhalterkomponenten simuliert werden, daß ein lauffähiger Prototyp entsteht, dessen Kernfunktionalität vollständig fertiggestellt wurde, dessen periphere Funktionalität aber beschränkt ist.
Es mußten folgende Arbeiten erledigt werden:

- Implementierung der Phase-2 Werkzeugkomponente 'STD-Assistant'.
  Diese Komponente steuert und verwaltet sämtliche unter- und zugeordneten Phase-1 Editoren und Phase-2 Hilfswerkzeuge. Diese Komponente stellt das eigentlich Werkzeug zur Bearbeitung von Automatendokumenten dar.

- Übernahme von Pflege, Anpassungs- und Erweiterungsarbeiten der folgenden Komponenten:

  - FriscoDiagramEditor zum Bearbeiten von AutomatenDiagrammen
  - SimpleTableEditor zum Bearbeiten von Zustands und Transitionstabellen
  - FriscoF, einer Komponente, die einen Parser für die funktionale Sprache FriscoF bereitstellt. Das zugehörige Paket enthält den Parser in Form einer Java-Bibliothek.

  Die Komponente SimpleTextEditor befindet sich auf dem neusten Stand und konnte ohne Erweiterung benutzt werden.

- folgende Pakete mußten neu erstellt werden:
  - Proof-Manager. Eine Komponente zum Verwalten von Beweisverpflichtungen. Die Benutzerschnittstelle war so zu gestalten, daß zu einem späteren Zeitpunkt ein Beweiser angeschlossen werden kann.

Die grundlegenden Schwierigkeiten lagen zunächst im reinen Umfang der Aufgabenstellung und der Art der Realisierung, d.h. als System teilweise eigenständiger und nicht für diesen Einsatzzweck optimierter, d.h. wiederverwendbarer Komponenten. Zudem mußte für jede zu entwickelnde Komponente eine offene und modulare Architektur entworfen werden, die es erlaubt, auch nachträglich spezifizierte Funktionen zu implementieren, oder Teile der Komponente auszutauschen, ohne die Komponente vollständig zu erneuern.
Eine weitere Schwierigkeit war die Umsetzung der theoretischen Automatendokumente in ein reales Werkzeug. Zwar wurde die Theorie eigens dafür entworfen, in ein Werkzeug implementiert zu werden, jedoch ist das Verfahren, ein Dokument zur Eingabe von Information, also als Benutzerschnittstelle zu verwenden ein schwieriges Unterfangen, da eine dokumentenorientierte Darstellung naturgemäß auf ganz andere Bedürfnisse hin optimiert ist, als auf eine leichte Implementierung in ein Computerprogramm. Die Darstellung ist vielmehr ganz und gar auf menschliche Bedürfnisse hin ausgerichtet, nämlich Übersichtlichkeit und redundanzfreie, intuitive Eingabe.
Besonders die Redundanzfreiheit stellte relativ hohe Anforderungen an die Implementierung. Für einen Menschen sind viele Informationen implizit gegeben, die für ein Computerprogramm nur mit ungleich höherem Aufwand zugänglich sind.

4.3. Das Kernsystem

4.3.1. Aufgaben des Kernsystems

Folgende Aufgaben fallen in die Zuständigkeit des Kernsystems (siehe auch Abbildung 4.2):

- Kontrolle und Steuerung eines Automatendokumentes. Dazu gehört die Verwaltung ...
  - eines Diagramms, das die graphische Darstellung eines Automaten enthält.
  - einer Tabelle, die weitere Informationen über die einzelnen Zustände des Automatendiagramms enthält.
  - einer Tabelle, die weitere Informationen über die einzelnen Transitionen des Automatendiagramms enthält.

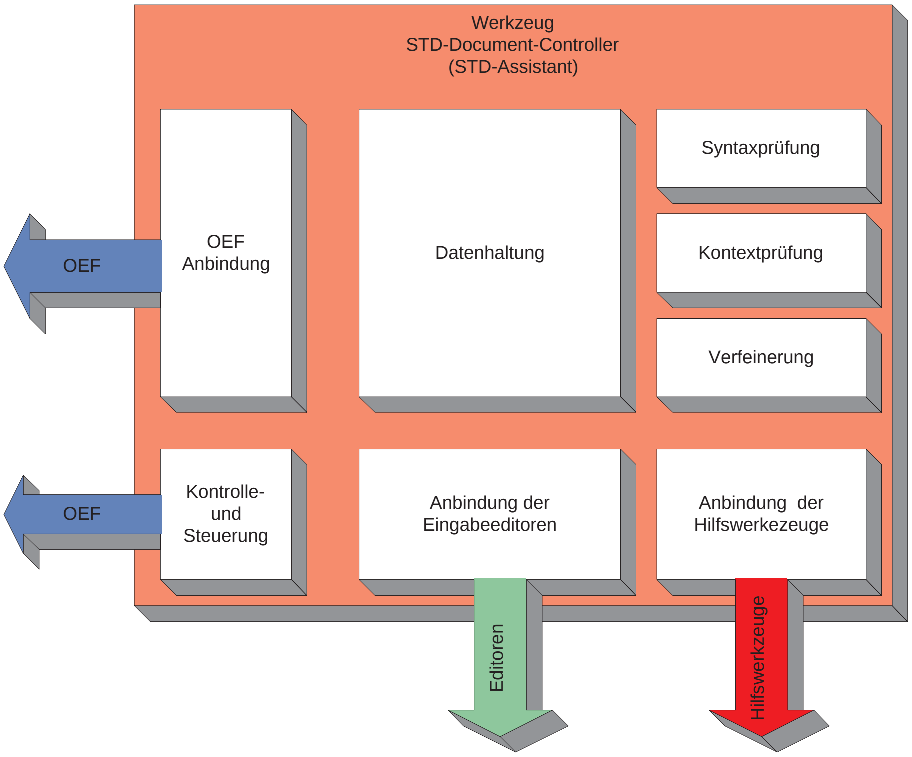


Abbildung 4.2.: Aufgaben des Dokumentenkontrollers

  - einem Textfeld, das sonstige, für das Automatendokument als ganzes gültige Informationen enthält.
  - eines Hilfswerkzeuges, das die Verwaltung von Beweisverpflichtungen ermöglicht.
  - einer Kontroll- und Steuertafel, mit der die Hauptfunktionen gesteuert werden und über die Kontrollinformationen, beispielsweise unkritische Fehlermeldungen, Warnungen und Hinweise ausgegeben werden können.

- Synchronisation aller Ein- und Ausgabedaten sowie deren gesamtheitliche Erfassung in einer zentralen Datenstruktur dergestalt, daß sich dem Benutzer das gesamte Automatendokument als Model-View-Mechanismus präsentiert, d.h. die verschiedenen Editoren als Ansichten (eines Ausschnittes) ein und derselben Information aufgefaßt werden können, wobei die Information unterschiedlich repräsentiert sein kann (z.B. einmal als Text, einmal als Grafik).

  Tatsächlich ist der Grad der Überschneidung sehr gering, so daß diese Lösung zunächst als zu kompliziert erscheint, sie wird aber hauptsächlich durch die in Abschnitt 4.2 beschriebene Notwendigkeit zur Wiederverwendbarkeit und Austauschbarkeit der Eingabeeditoren und den Umstand induziert, daß komplexe interne Operationen eine gemeinsame und synchronisierte Datenbasis voraussetzen (siehe auch die folgenden Punkte).

- Implementierung einer syntaktischen Prüfung. Da die Eingabe von Daten mittels semantikfreier Editoren stattfindet, die (um möglichst wiederverwendbar zu sein) keine speziellen syntaktischen Einschränkungen bei der Eingabe implementieren, müssen die syntaktischen Besonderheiten des Automatendokumentes in einem separaten Modul des Kernsystems geprüft, oder deren Prüfung veranlaßt werden.

- Implementierung einer Kontextprüfung. Die Theorie des Automatendokumentes impliziert einige Kontextbedingungen, vor allem durch die Verwendung einer funktionalen Sprache.

  Die Kontextprüfungen müssen in einem separaten Modul des Kernsystems erledigt, oder deren Prüfung veranlaßt werden, beispielsweise durch Übergabe an einen Parser.

- Implementierung der Verfeinerung für Automatendokumente. Dazu gehört:

  - Prüfung der Korrektheit des Automatendokumentes.
  - Einschränkung der Eingabemöglichkeiten auf festgelegte interaktive Verfeinerungsaktionen.
  - Erneute Korrektheitsprüfung nach jedem Verfeinerungsschritt.

  Die Prüfung der Korrektheit erfolgt durch die Ausgabe sogenannter Beweisverpflichtungen. Können die Beweisverpflichtungen erfüllt werden, ist auch die Korrektheit des Automatendokumentes bzgl. der Bedingungen KB und VB bewiesen. Dazu werden die Bedingungen an eine separate Komponente zur Verwaltung von Beweisverpflichtungen übergeben.

- Skriptsteuerung. Es wurde besonderen Wert auf die Möglichkeit gelegt, während der Bearbeitung des Automatendokumentes ein Skript, d.h. eine textbasierte Auflistung aller Arbeitsschritte zu erstellen, das nach Möglichkeit wiederverwendbar ist, d.h. auch auf andere Automatendokumente angewandt werden kann, um dort die gleichen Konstruktionsschritte anzuwenden.

- Weiterverarbeitung. Es wurde Wert darauf gelegt, ein fertiges Automatendokument geeignet zu exportieren, um es an anderer Stelle außerhalb des OEF zu verwenden, beispielsweise in schriftlichen Dokumentationen.

4.3.2. Funktionen

Folgende Funktionen waren im einzelnen zu implementieren:

- Aktivieren/Deaktivieren von im Hintergrund ausgeführten Syntaxprüfungen.
- Aktivieren/Deaktivieren von im Hintergrund ausgeführten Kontextprüfungen.
- Aktivieren der Verfeinerung.
- Durchführen eines kompletten Syntaxchecks für das ganze Dokument.
- Durchführen eines kompletten Kontextchecks für das ganze Dokument.
- Durchführen einer kompletten Korrektheitsprüfung für das ganze Dokument, als Vorbereitung für die Aktivierung der Verfeinerung.

Desweiteren eine Unterstützung der folgenden OEF-Funktionen:

- Scripting
- Undo/Redo
- Ausgabe in Latex als Druckvorstufe.
- Annotationen zur Anzeige von Hinweisen und Fehlermeldungen direkt in den Eingabeeditoren.

Während der Verfeinerung sind folgende Funktionen verfügbar:

- Hinzufügen neuer Zustände (Anwendung der Regel **addS**)
- Entfernen von Zuständen (Anwendung der Regel **remS**)
- Verfeinern eines Zustandes (Anwendung der Regel **refS**)
- Hinzufügen neuer Transitionen (Anwendung der Regel **addT**)
- Entfernen von Transitionen(Anwendung der Regel **remT**)
- Verfeinern einer Transition(Anwendung der Regel **refT**)
- Entfernen von Initialtransitionen(Anwendung der Regel **remI**)
- Verfeinern einer Initialtransition(Anwendung der Regel **refI**)

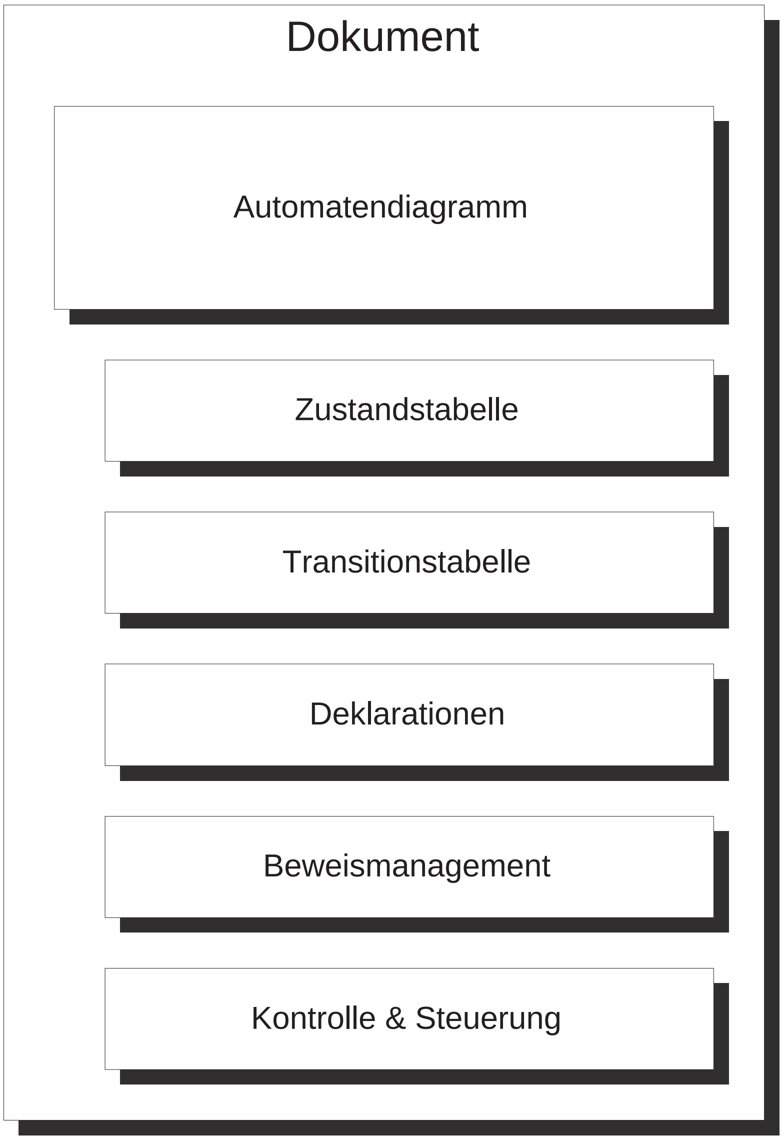


Abbildung 4.3.: Layout des Automatendokumentes unter OEF

4.3.3. Darstellung

Das Dokumentenlayout

Das Dokumentenlayout (Abb. 4.3) wurde nach den Richtlinien für OEF-Phase-2 Werkzeugen konstruiert, d.h. die Kontroll- und Steuertafel für das Werkzeug befindet sich ganz unten im Dokument. Darüber befinden sich eventuelle Hilfswerkzeuge. Editoren und Viewer befinden sich ganz oben im Dokument.

Kontroll- und Steuertafel des Hauptwerkzeuges

Die Kontroll- und Steuertafel des Hauptwerkzeuges enthält Möglichkeiten zur Aktivierung der wichtigsten Funktionen (Syntax-, Kontext- und Korrektheitsprüfung sowie Aktivierung der Verfeinerung), zur Eingabe eines Klassenbezeichners und Typs des Automatendokumentes sowie ein Listenfeld zur Anzeige eines momentanen Status (aktuelle Fehler- und Warnhinweise).

4.3.4. Steuerung

Im folgenden wird nur die Steuerung der Benutzerschnittstelle des Kernsystems beschrieben. Die zur Ein- und Ausgabe verwendeten Editorkomponenten besitzen eine eigene Benutzerschnittstelle, gleiches gilt für verwendete Hilfswerkzeuge. Außerdem ist zu bedenken, daß alle Benutzerschnittstellen in die Benutzerführung der Laufzeitumgebung OEF integriert und teilweise eng mit dieser verzahnt sind.
Viele implementierte Funktionen werden direkt von der Laufzeitumgebung unterstützt und deshalb auch über deren Benutzerschnittstelle bedient.

Maussteuerung

- Die Funktionen zum (De-)Aktivieren von Hintergrundprüfungen und der Verfeinerung können über Checkboxen aktiviert werden. Ein Klick auf eine freie Checkbox aktiviert die jeweilige Funktion, ein Klick auf eine markierte Checkbox deaktiviert die Funktion. Da eine Überprüfung des Kontexts erst nach erfolgreicher Prüfung der Syntax Sinn macht, bzw. die Verfeinerung nur durchgeführt werden kann, wenn gleichzeitig Syntax und Kontext überprüft werden, führt eine Aktivierung von Kontext oder Verfeinerung auch gleichzeitig zur Aktivierung der vorausgesetzten Funktionen. Eine Deaktivierung der Verfeinerung ist laut Vorgabe nicht möglich.

- Überprüfung des gesamten Automatendokumentes auf Syntax, Kontext und Korrektheit ist auch fallweise möglich. Dazu muß einer der entsprechend beschrifteten Knöpfe gedrückt werden.

- Ein Klick auf einen Hinweis (z.B. Fehlermeldung) im Textfeld des Steuerpaneels führt je nach Fähigkeit der betroffenen Editoren zu einer farbigen Hervorhebung des betroffenen Elementes.

Werkzeugleiste

Neben variierenden Funktionen, die davon abhängen, welcher Teil des Automatendokumentes gerade bearbeitet wird, müssen bei aktivierter Verfeinerung alle Verfeinerungsoperationen über die Werkzeugleiste erreichbar sein.

4.3.5. Weitere nichtfunktionale Anforderungen

4.3.5.1. Sprache

Es galten folgende Mindestanforderungen: Die Sprache des Programms, des Quelltextes und der Onlinehilfe ist Englisch. Die Sprache des Handbuches und der schriftlichen Dokumentation darf auch Deutsch sein. Zum Programmieren ist die momentan aktuelle Version der Sprache Java und der JFC[1] zu verwenden.

4.3.5.2. Hilfe

Als Mindestanforderung galt das Vorhandensein eines Benutzerhandbuchs im HTML[2]-Format, einer API[3]-Beschreibung im HTML-Format und einer gedruckten Dokumentation des gesamten Pakets. Die Elemente der Werkzeugleiste mußten mit 'Tooltips' versehen werden, einer kurzen Beschreibung, die eingeblendet wird, wenn der Benutzer zögernd mit dem Mauszeiger über einem Steuerelement verharrt.

4.3.5.3. Performance

Aufgrund der besonderen Situation, in der sich in Java realisierte Projekte momentan befinden, gab es an dieser Stelle nur geringe Vorgaben: Die Ausführungsgeschwindigkeit muß ausreichend sein, um ein Arbeiten mit dem Produkt möglich zu machen.

4.3.5.4. Testfälle

Als Testfall galt die Erstellung des Tutorials und aller anderen in dieser Arbeit verwandten Beispiele.

4.4. Periphere Komponenten

Sämtliche notwendigen und bereits existierenden peripheren Komponenten mußten so weit angepaßt werden, daß die Systemanforderungen erfüllt werden konnten. Alle nicht existierenden Komponenten mußten simuliert, oder so weit im-

[1] Java Foundation Classes
[2] HyperText Markup Language
[3] Application Programming Interface

provisiert werden, daß letztendlich eine aussagekräftige Demonstration des Werkzeuges möglich ist.

4.4.1. Diagrammeditor

Der existierende Diagrammeditor FDE[4] mußte um Druckfunktionen und die Möglichkeit, Symbole zeitweise farbig hervorzuheben erweitert werden.

4.4.2. Tabelleneditor

In Ermangelung einer angemessen schnellen Tabellenkomponente mußte der Tabelleneditor improvisiert werden.

4.4.3. Werkzeugübergreifende Datenübernahme

In Ermangelung von Werkzeugen zur Bearbeitung der anderen Dokumentarten wurde die Ergebnisübergabe durch eine Textkomponente simuliert, in die benötigte Informationen eingetragen werden können.

4.4.4. Sprachunterstützung - FriscoF

Die ursprünglich geplante Anpassung des vorhandenen Parsers an die Erfordernisse von NLTF[5] (siehe auch Abschnitt 5.1.5) konnte aufgrund des Umfangs der notwendigen Änderungen nicht im Rahmen dieser Arbeit durchgeführt werden. Statt dessen wurde der Parser ohne größere Änderungen übernommen und die wichtigsten NLTF Parserfunktionen wurden in das Sprachunterstützungsmodul des Hauptwerkzeuges so weit integriert, daß eine Vorführung möglich ist.

4.4.5. Proof-Manager

Diese Komponente wurde vollkommen neu erstellt. Allerdings wurden nur die unbedingt notwendigen Basisfunktionen implementiert. Ein Anschluß eines Beweisers ist vorgesehen, kann allerdings erst erfolgen, wenn ein in Java geschriebener Beweiser vorliegt.

4.5. Erweiterungsmöglichkeiten

Die folgenden Punkte wurden während des Systementwurfes berücksichtigt, werden aber erst zu einem späteren Zeitpunkt implementiert:

- Anschluß eines Quelltextgenerators.

[4] Frisco Diagram Editor
[5] Nichtlineare-(Quell-)Textfelder

- Anschluß eines automatischen Beweiser, sobald vorhanden (siehe auch Abschnitt 4.4.5).
- Anschluß eines NLTF-Parsers.

4.6. Zusammenfassung

4.6.1. Besondere Leistungsmerkmale

Die Spezifikation des Automatenwerkzeuges enthält einige Besonderheiten, die dieses Werkzeug von anderen unterscheiden:

- Skriptsteuerung

  Sämtliche Arbeitsschritte lassen sich zur späteren Wiederverwendung als Skript aufzeichnen. Dies kann sinnvoll sein, wenn bestimmte Entwicklungsschritte in Form einer Vorlage immer wieder angewendet werden sollen, oder bei der Konstruktion in einem frühen Stadium ein Fehler gemacht wurde. Dann können wesentliche Schritte, soweit anwendbar, schnell automatisiert nachvollzogen oder wiederholt werden, ohne daß die gesamte Konstruktion erneut per Hand durchgeführt werden muß.

- Offene Komponentenarchitektur

  Die Tatsache, daß auf vorhandene Editoren zur Eingabe von Informationen zurückgegriffen wird, spart zunächst einmal Zeit, die allerdings wurde von der Entwicklung und Implementierung einer entsprechend komplizierten Architektur fast vollständig aufgezehrt.

  Ein wirklicher Vorteil entsteht erst dadurch, daß die Architektur einerseits auch für andere Werkzeuge verwendet werden kann, andererseits in der Theorie immer nur ein Editor für jede Informationsdarstellung benötigt wird. Dies senkt die Wartungskosten ganz erheblich. Zudem profitieren bei einer Verbesserung dieses einen Editors automatisch alle Werkzeuge, die den nun verbesserten Editor verwenden. Weiterhin ist dieses Verfahren ressourcenfreundlich, weil ein Großteil des Programmcodes so von mehreren Werkzeugen geteilt wird und benutzerfreundlich, weil der Anwender sich in den meisten Werkzeugen schnell zurecht findet, da immer die gleichen Editoren verwendet werden.

  Natürlich kann es in der Praxis sinnvoll sein, für jede Informationsart mehr als einen Editor anzubieten, da beispielsweise nicht jedes Werkzeug, das eine Tabelle verwendet, gleich eine komplette Tabellenkalkulation benötigt; das Gesagte bleibt aber prinzipiell richtig.

- Verfeinerung

  Das vorliegende Werkzeug dürfte eines der ersten Werkzeuge sein, das einen mathematisch fundierten Verfeinerungskalkül implementiert und damit die Möglichkeit bereitstellt, schon während der Spezifikation eines Softwareprojektes automatisierte semantische Überprüfungen durchzuführen.

- Annotationen

  Fehlermeldungen werden nicht nur gesammelt an einer Stelle aufgelistet, wobei die Zuordnung sehr schwierig sein kann, vor allem da ein Automatendokument keine Zeilennummern kennt, sondern sie werden auch direkt in den jeweiligen Editoren durch sogenannte Annotationen markiert.

  Eine Annotation läßt sich für diesen Anwendungszweck am ehesten mit einer Fußnote vergleichen. Annotationen werden direkt von der Laufzeitumgebung OEF unterstützt.

5. Systembeschreibung

Dieses Kapitel enthält eine ausführliche technische Beschreibung des Werkzeuges STDA. Ausgehend von den in Kapitel 4 beschriebenen Anforderungen wird in Abschnitt 5.1 zunächst der Entwurf des Gesamtsystems erläutert und wichtige Kernprobleme isoliert. Ab Abschnitt 5.1.3 werden Lösungen für die interessantesten Kernprobleme erarbeitet und vorgestellt. Aus Gründen der Übersichtlichkeit, und um technisch interessierten Anwendern entgegenzukommen, wurden diese Abschnitte, die eigentlich thematisch stärker der Beschreibung der Einzelkomponenten zugeordnet sind, extrahiert und an einer Stelle gesammelt wiedergegeben.
Mit Abschnitt 5.2 beginnt dann die Einzelbetrachtung der verschiedenen Komponenten des Gesamtsystems. Diese ist stärker technisch orientiert als der Beginn des Kapitels und richtet sich vor allem an Programmierer, die Wartungsarbeiten, Änderungen oder Erweiterungen vornehmen wollen und eine detailliertere Beschreibung benötigen.
Es ist zu bedenken, daß nur ein kleiner Ausschnitt der Werkzeugimplementierung beschrieben werden kann, da der Umfang dieser Arbeit sonst übermäßig zunehmen würde. Außerdem wäre die Beschreibung sonst schon nach wenigen Wochen veraltet.

5.1. Das Gesamtsystem

5.1.1. Systemarchitektur

Die Zerlegung des Gesamtsystems in Teilsysteme ist größtenteils bereits durch die Systemanforderungen in Kapitel 4 vorgegeben. Der Aufbau des Systems erfolgt weitestgehend gemäß den Richtlinien zur Erstellung von Phase-2-Werkzeugen für OEF, d.h. Aufspaltung in mehrere semantikfreie Phase-1-Editoren, Phase-2-Hilfswerkzeuge und eine Steuer-/Kontrollkomponente. Diese Komponente kümmert sich um die Verwaltung der benötigten Viewer und Editoren und deren Anordnung innerhalb eines Dokumentes. Aus diesem Grund wird diese Komponente fortan *Documentcontroller* genannt. Der Documentcontroller ist das Herzstück der Werkzeuganwendung, in dem alle Fäden zusammenlaufen. Der Documentcontroller kann sich selbst im kontrollierten Dokument mittels eines Kontroll- und Steuerpaneels anzeigen, über das zusätzliche Funktionen des

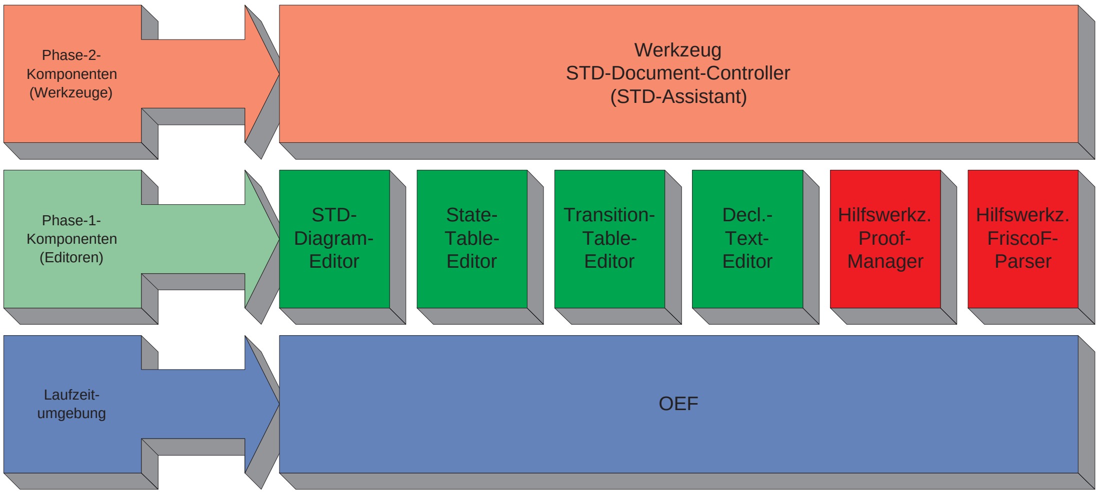


Abbildung 5.1.: Gesamtarchitektur - Zerlegung in Teilsysteme

Werkzeuges bedient werden können, die nicht über einen der Editoren oder eines der Hilfswerkzeuge erreicht werden können.
Der Documentcontroller erledigt, falls nötig, ebenfalls die Kommunikation mit anderen Phase-2-Werkzeugen über die Dokumentgrenzen hinweg.

Zusammenfassung der Aufgaben des Documentcontrollers

- **Organisation und Management**
  - Kontrolle über das Hinzufügen und Löschen von Viewern, Editoren und Hilfswerkzeugen.
  - Layoutkontrolle des verwalteten Dokumentes.
- **Werkzeugsteuerung**
  - Implementiert die Funktionalität, die über die Ein- und Ausgabe semantikfreier Information hinausgeht und nicht von Hilfswerkzeugen behandelt wird.
  - Implementiert eine Benutzerschnittstelle, welche die Steuerung des gesamten Werkzeuges ermöglicht.
- **Kommunikation**
  Der Documentcontroller dient als Kommunikationszentrale. Er kommuniziert mit
  - den von ihm verwalteten Phase-1-Komponenten zur Eingabe oder zur Anzeige von Information.
  - den von ihm verwalteten Phase-2-Hilfswerkzeugen, zur Erledigung von Grundaufgaben, die auch in anderen Werkzeugen häufig genutzt werden, beispielsweise das Parsen einer Sprache.

  - anderen Werkzeugen für eine dokumentübergreifende Organisation, beispielsweise zur Übernahme von Ergebnissen aus anderen Werkzeugen.

Eine dokumentübergreifende Kommunikation ist beim momentanen Stand der Entwicklung nicht vorgesehen und wird dadurch simuliert, daß benötigte Informationen, die aus anderen Dokumenten übernommen werden könnten, konkret handelt es sich dabei um die Liste der Typen, Attribute, Ein- und Ausgabemethoden, an entsprechender Stelle eines Editors neu eingegeben werden müssen.
Unter Berücksichtigung aller gegebenen Anforderungen und Richtlinien, vor allem des in den Systemanforderungen (siehe Kapitel 4) beschriebenen Prinzips der Wiederverwendbarkeit und Austauschbarkeit sowie bereits vorhandener Komponenten, entsteht folgende Komponentenhierarchie:

- Werkzeug & Documentcontroller: STD-Assistant
- Phase-1-Editoren zur Ein-/Ausgabe:
  - Automatendiagramm: FriscoSTDEditor
  - Zustands & Transitionstabellen: SimpleTableEditor
  - Deklarationen: SimpleTextEditor
- Phase-2-Hilfswerkzeuge:
  - Verwaltung von Verifikationsbedingungen: ProofManager
  - Parser für funktionale Sprache: FriscoF
    FriscoF ist momentan nur eine Programmbibliothek und kein vollständiger Part. Der Parser wird in der Architektur aber als Part modelliert, da abzusehen ist, daß der letztendlich zum Einsatz kommende NLTF-Parser eine Kontrollkonsole besitzen wird und damit ein vollständiger Parthandler sein wird.

5.1.2. Kommunikationsfluß

Die Kommunikation zerfällt in zwei Kategorien:

- **Horizontale Kommunikation:**
  Hierunter ist die Kommunikation zwischen Werkzeug, Hilfswerkzeugen und Editoren zusammengefaßt. Das Werkzeug kontrolliert alle anderen Komponenten, tauscht Informationen mit diesen aus, oder ruft Funktionen auf.
- **Vertikale Kommunikation:**
  Vertikale Kommunikation ist jedwede Kommunikation mit der Laufzeitumgebung. Aufrufe von Funktionen der Umgebung, oder Funktionsaufrufe von der Umgebung an die einzelnen Komponenten.

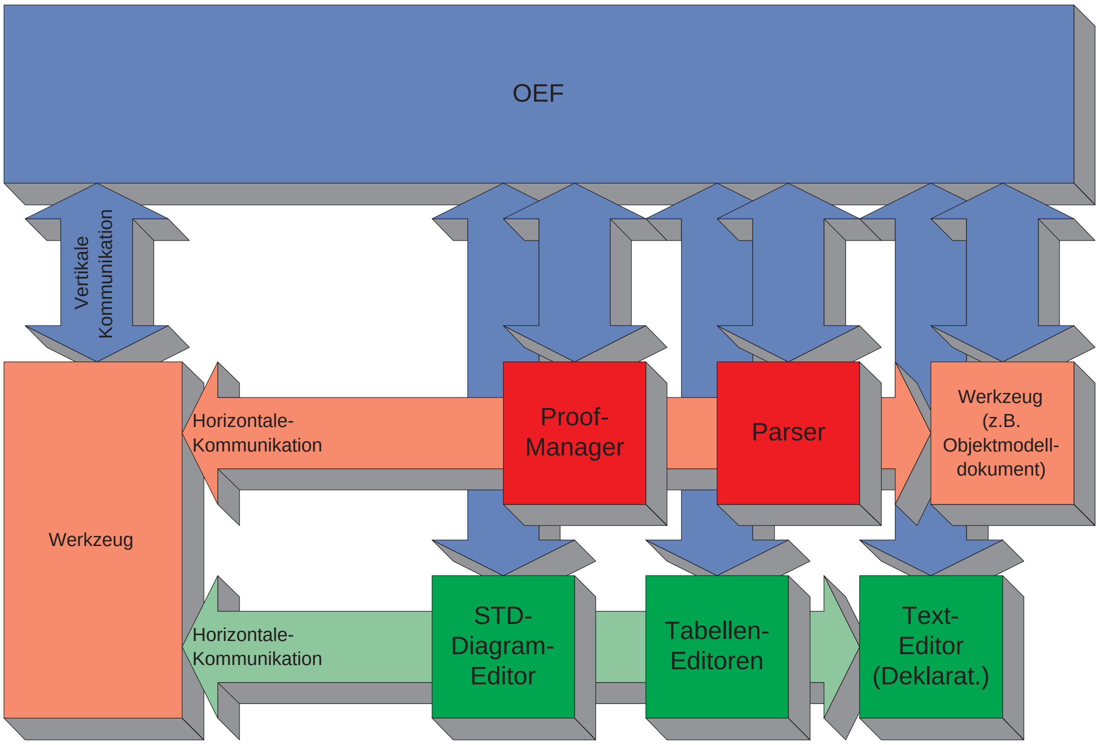


Abbildung 5.2.: Kommunikation - horizontal und vertikal

Dabei sind die Möglichkeiten der Hauptkomponente des Werkzeuges, in die Kommunikation von Editoren und Hilfswerkzeugen mit der Laufzeitumgebung einzuwirken stark beschränkt. Im wesentlichen existieren dafür nur zwei Möglichkeiten:

1. Indirekt über das Interface der Komponente. Falls die Komponente ein Interface bereitstellt, das es erlaubt, ihre Kommunikation mit der Laufzeitumgebung zu konfigurieren, beispielsweise ob ausgeführte Aktionen an die Skriptaufzeichnung, oder den Undo-Stack gemeldet werden, oder nicht.

2. Direkt über Modifikation des Scriptinterface. Falls die Komponente eine Möglichkeit bereitstellt, ihr Scriptinterface zu modifizieren, kann auf diese Weise ein Filter im Hauptkommunikationskanal mit der Laufzeitumgebung installiert werden. Wie gut das funktioniert, ist stark von der Implementierung der jeweiligen Komponente abhängig.

Im folgenden findet sich eine detailliertere Beschreibung der interkomponentären Kommunikation. Die Kommunikation zwischen Komponenten des Gesamtsystems und der Laufzeitumgebung kann in der Systembeschreibung der jeweiligen Komponente nachgelesen werden:

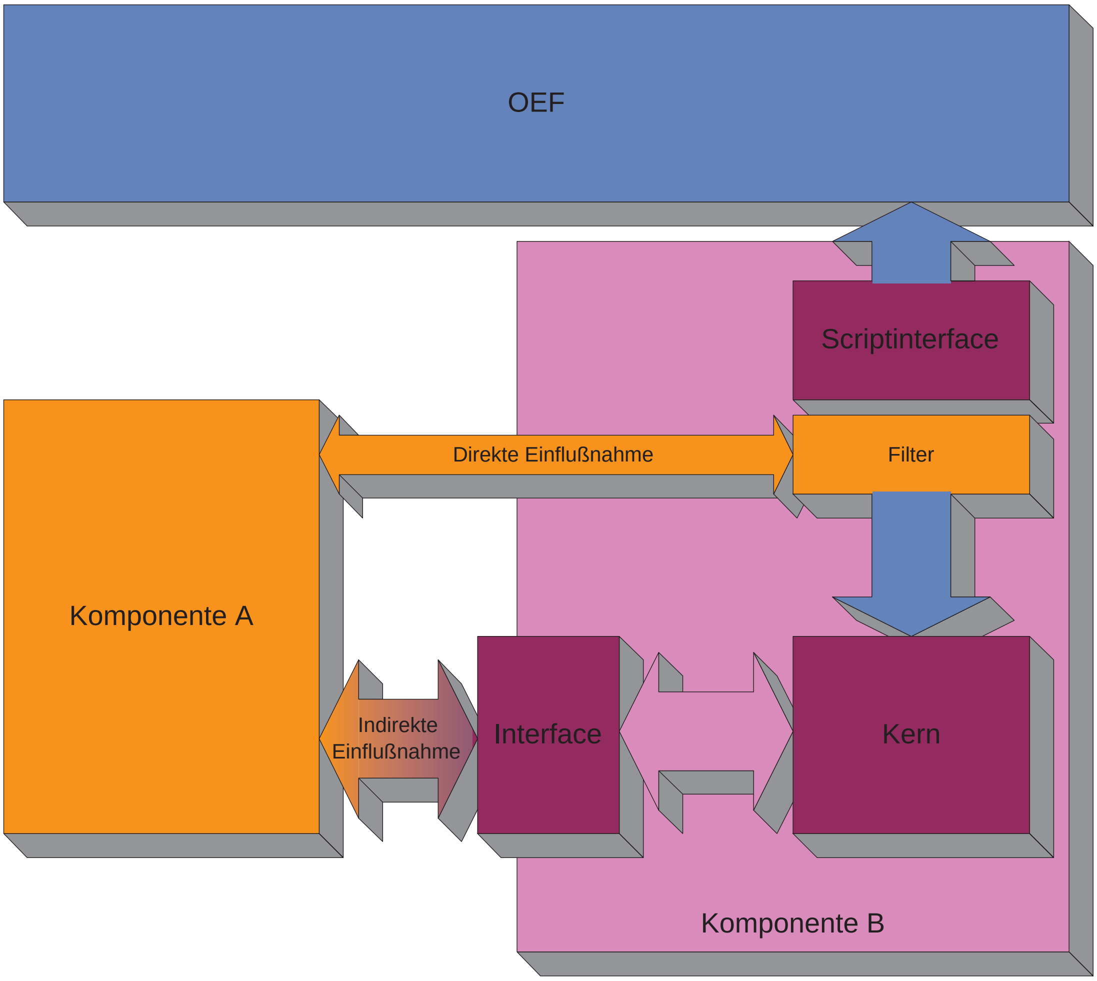


Abbildung 5.3.: Einflußnahme auf die vertikale Kommunikation

Komponente	**Beschreibung der Kommunikation mit der Laufzeitumgebung**
STD-Assistant	siehe Abschnitt 5.2.2
FriscoSTDDiagramEditor	siehe [6]
SimpleTextEditor	siehe [30]
SimpleTableEditor	siehe [30]
Proof-Manager	siehe 5.3
FriscoF	siehe [15]

5.1.3. Der SDMV[1] Mechanismus

Die besonderen Anforderungen, die an dieses Projekt gestellt wurden, machten die Entwicklung eines erweiterten Model-View-Mechanismus notwendig. Da ein Verständnis dieses Mechanismus essentiell für das Verständnis der Architektur des Gesamtsystems ist, wird ihm an dieser Stelle ein eigener Abschnitt gewidmet.

5.1.3.1. Motivation

Um zu verstehen, wodurch SDMV motiviert wird, muß man zunächst die Philosophie verstehen, die hinter der Laufzeitumgebung OEF steht.
Im Gegensatz zu anderen Softwareentwicklungswerkzeugen, wie beispielsweise Autofocus wurde das OEF von vornherein nicht als Werkzeug, sondern als *Framework* und als *Laufzeitumgebung* für Werkzeuge entwickelt, d.h. es handelt sich um eine Werkzeugplattform. Diese Plattform stellt also einerseits eine API zur Verfügung, welche Funktionen enthält, die allen Softwareentwicklungswerkzeugen gemeinsam sind, wie beispielsweise die Verwaltung eines Undo-Stacks, Elementen für die Benutzerschnittstelle usw.
Zum anderen besteht diese Plattform aus einer Rahmenapplikation, die als Ablaufumgebung für die einzelnen Werkzeuge fungiert. Diese Rahmenapplikation implementiert sämtliche Funktionalität, die zur Verwaltung, Organisation und zur Darstellung von Werkzeugkomponenten und deren Benutzerschnittstellen notwendig ist.
Ein Werkzeugentwickler profitiert von dieser Konstruktion, indem die von ihm programmierten Werkzeuge nur die eigentliche Werkzeugfunktionalität enthalten müssen, alles andere kann durch relativ einfache API-Aufrufe erledigt werden. Des weiteren ist dieses Verfahren ressourcenschonender, ein nicht unwichtiger Umstand bei Projekten, die in Java realisiert werden. Ressourcenschonender, weil dadurch ähnliche Funktionen in unterschiedlichen Werkzeugen nicht doppelt implementiert werden müssen und weil diese dann bei der Ausführung nicht doppelt im Speicher stehen, ressourcenschonender, weil die freiwerdenden Kapazitäten in

[1] Synchronized-Data-Model-View

die Optimierung und Verbesserung der einen Implementierung gesteckt werden können und zuletzt, weil notwendige Wartungsarbeiten nur einmal durchgeführt werden müssen und nicht mehrfach in jedem einzelnen Werkzeug.
Der Anwender dagegen profitiert von dieser Konstruktion, indem sich ihm alle Werkzeuge mit einer weitgehend einheitlichen Benutzerschnittstelle präsentieren. Egal, welches Werkzeug er gerade benutzt, die Undo-Funktion läßt sich immer auf die gleiche Weise aktivieren.
Soweit nichts aufregend neues möchte man sagen. Beim Entwurf des OEF wurde jedoch, basierend auf den Erkenntnissen der obigen Absätze, noch weiter nachgedacht. Was wäre, wenn man das Prinzip von ein wenig ausweiten würde, sich nicht nur auf gemeinsame Basisfunktionen und einfache Elemente der Benutzerschnittstelle beschränkt, sondern diesen Ansatz konsequent zu Ende denkt und auch Funktionalitäten höherer Ordnung, die in mehr als einem Werkzeug Verwendung finden könnte, so zu gestalten, daß sie mehrfach verwendbar sind. Es entstand der Gedanke, weniger komplette Werkzeuge zu entwickeln, als vielmehr eine Art Baukasten, der aus möglichst einfachen und allgemein anwendbaren Modulen besteht, aus denen leicht komplexe Werkzeuge zusammengesetzt werden könnten.
Auf diese Weise könnte man die gleichen Vorteile, die zunächst nur für das Prinzip einer Werkzeugplattform galten, auch auf die höhere Ebene der Werkzeuge retten und so noch mehr davon profitieren. Der Gedanke ist zugegebenermaßen nicht ganz neu und stellt angesichts monolithischer Anwendungen, an denen nicht nur der Anwender, sondern auch die entwickelnden Firmen langsam verzweifeln, nur eine logische Evolution in der Softwareentwicklung dar.

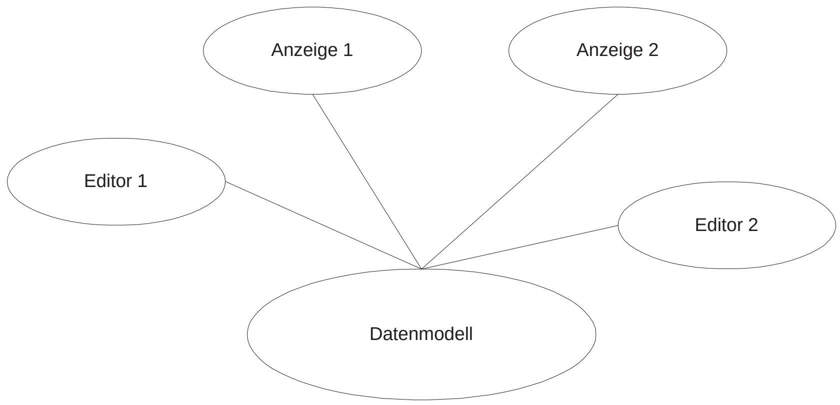


Abbildung 5.4.: Model-View-Mechanismus

Als vornehmliche Kandidaten für die Wiederverwendung kommen jegliche Art von Editoren oder Viewern in Frage. Viele Werkzeuge benutzen beispielsweise auf die eine oder andere Weise Diagramme, längere Texte oder Tabellen. Es lag nun nahe, diese Editoren einmal als Komponente zu entwickeln und dann immer wieder in verschiedenen Werkzeugen zu verwenden.

Die Editoren müssen natürlich so programmiert werden, daß sie in möglichst vielen Werkzeugen verwendet werden können. Das bedeutet aber, daß die Editoren nur eine (für den Editor) semantikfreie Repräsentation der Information darstellen können.
Zudem benötigt jeder Editor natürlich ein eigenes semantikfreies Datenmodell, um die Informationen zu speichern. Dies bedeutet aber, daß eine Komponente, die dann letztendlich eine gewünschte Werkzeugfunktionalität implementiert, dann schlußendlich, falls sie einen dieser semantikfreien Editoren benutzt, irgendeine Beziehung zwischen dem eigenen Datenmodell, das mit hoher Wahrscheinlichkeit eine Semantik besitzt, und dem des verwendeten Editors, das ja außer der Semantik, die zur Repräsentation der Information unbedingt notwendig ist, keine höhere Semantik besitzt, herstellen muß.
Falls nun auch auf der Datenbasis des Werkzeuges Operationen durchgeführt werden, die diese verändern, dann muß diese Beziehung in beide Richtungen ausgelegt werden. Die Datenmodelle der beiden Komponenten müssen also *synchronisiert*

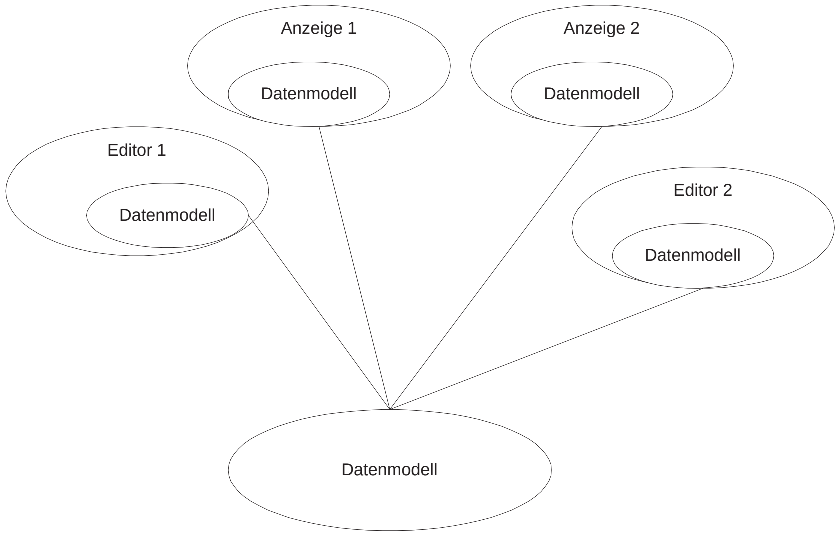


Abbildung 5.5.: Erweiterter Model-View-Mechanismus

werden. Für den Benutzer bleibt dieser Vorgang verborgen (im amerikanischen Sprachgebrauch transparent). Für ihn präsentiert sich das Zweigespann der Komponenten lediglich als eine bestimmte Ansicht (view) auf ein Datenmodell (datamodel), da sich die synchronisierten Datenbasen wie ein einziges Datenmodell darstellen.
Dies ist auch schwieriger zu implementieren als ein normaler Model-View Mechanismus, wo ja der Synchronisationsvorgang dadurch entfällt, daß tatsächlich auf eine gemeinsame Datenbasis zurückgegriffen wird.
Ungleich erschwert wird die Tatsache durch den Umstand, daß eine Anzeigekomponente, beispielsweise ein Editor, jederzeit in gewissen Grenzen durch eine andere Komponente ausgetauscht werden können soll. In gewissen Grenzen

natürlich, weil die ausgetauschte Komponente natürlich mindestens die gleiche Funktionalität und die gleiche Untermenge von Schnittstellen haben muß, wie die auszutauschende Komponente.

5.1.3.2. Beschreibung des SDMV-Mechanismus

Im letzten Abschnitt konnte man erkennen, daß der Unterschied zwischen SDMV und MV[2] nur darin liegt, daß die Selektion und Umformung in die gewünschte Repräsentationsform der anzuzeigenden Daten nicht während des Datenzugriffs im Anzeigemodul erledigt wird, sondern in einem früheren Schritt, durch den Abgleich einer zweiten Datenbasis, die dann die Daten bereits in einer für die Anzeige optimalen Form enthält. Umgekehrt gilt natürlich das gleiche. Die Daten werden nicht direkt nach der Eingabe konvertiert, sondern erst einmal zwischengespeichert und in einem zweiten Schritt konvertiert und erneut gesichert.

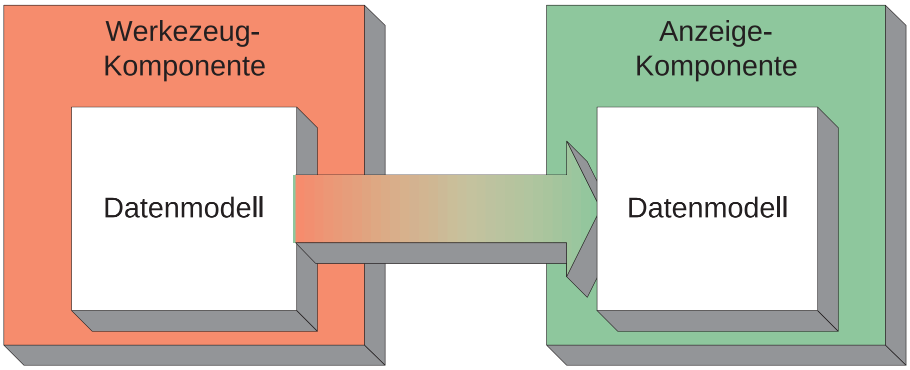


Abbildung 5.6.: SDMV: Primitive Lösung

Ein primitiver Ansatz zur Lösung des Problems könnte jetzt lauten, daß die Werkzeugkomponente einfach bei jeder Änderung ihres Datenzustandes diesen konvertiert und an die Anzeigekomponente übergibt. Bis auf den hohen Ressorcenverbrauch ist gegen diesen Ansatz nichts einzuwenden. Hat man mehr als eine Anzeige, oder eine größere Datenbasis, steigt der Aufwand natürlich unter Umständen schnell über eine Akzeptanzschwelle, bei Verwendung von Java eher früher als später. Die eklatanten Schwächen dieser Vorgehensweise treten aber vor allem dann zutage, wenn man die umgekehrte Richtung betrachtet. Handelt es sich nicht nur um eine Anzeige, sondern um einen Editor, dann müßte natürlich auch bei jeder Änderung der eingegebenen Information eine Konvertierung stattfinden. Diese Konvertierung kann ziemlich heikel werden, wenn der Editor nur einen Ausschnitt der Gesamtdatenbasis darstellt und bearbeitet. Bei einer vollständigen Konvertierung dürfen keine Informationen verlorengehen.

[2] Model-View

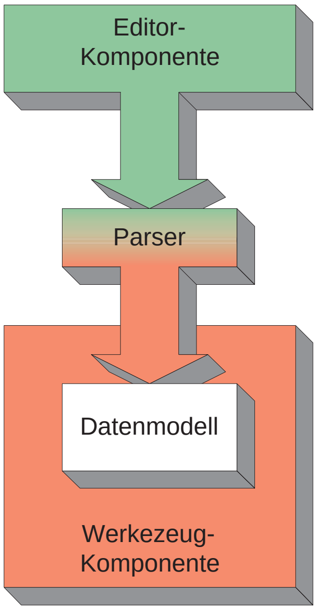


Abbildung 5.7.: SDMV: Eine andere Betrachtungsweise

Die ganze Problematik läßt sich auch noch ein wenig anders als mit einer Model-View-Betrachtungsweise formulieren. Man befindet sich in einer ähnlichen Situation wie ein Parser. Auch dieser muß semantikfreie Information, nämlich Text zunächst in einen abstrakten Syntaxbaum übersetzen, um die Semantik erfassen zu können. Der Parser hat den Vorteil, daß er nur in eine Richtung arbeiten muß und in der Regel keine weiteren Parser zum internen Datenmodell beitragen.
Anders sieht die Sache bei Werkzeugen für visuelle Programmierung aus. Hier hat man z.T. mehrere Editoren, die Daten beitragen und wünscht sich, daß sie in beide Richtungen funktionieren. Bei genauerer Betrachtung ist die zweite Richtung meist nur unsauber realisiert. Zudem wird nicht direkt auf eine gemeinsame Datenstruktur zugegriffen, sondern zuerst in ein Zwischenformat, nämlich den Quelltext übersetzt. Erst von dort wird dann in das abstrakte Datenmodell überführt. Für das vorliegende Problem ist diese Herangehensweise also nicht zu gebrauchen, obwohl sie als Denkvorlage dienen kann. Ein weiteres Problem ist die Handhabung einer Lösung, also wie einfach und übersichtlich sie dann schließlich und letztendlich programmiert werden kann. Eine vollständige Konvertierung wurde ja bereits verworfen, was aber bedeutet, daß bei jeder Änderung ein Abgleich derselben erfolgen muß:

- Bei einer Änderung der Gesamtdatenbasis müssen die Teildatenbasen der betroffenen Editoren und Anzeigen angeglichen werden.

- Bei der Änderung einer der Teildatenbasen, beispielsweise durch die Eingabe eines Benutzers, muß sowohl die Gesamtdatenbasis, als auch die Teilda-

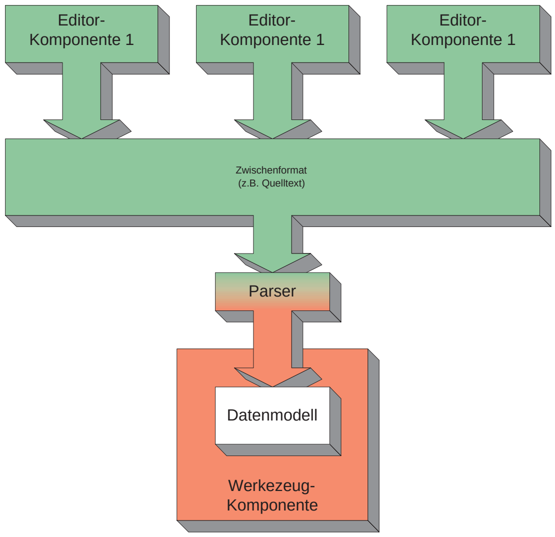


Abbildung 5.8.: SDMV: Ansatz visueller Programmierwerkzeuge

tenbasen evt. betroffener anderer Editoren und Anzeigen aktualisiert werden.

Bei jeder Änderung müßte also im Anschluß eine ganze Reihe von Operationen durchgeführt werden. Es ist einsichtig, daß dies nach Möglichkeit zentral erfolgen sollte, um den restlichen Code von Abgleichoperationen freizuhalten. Man

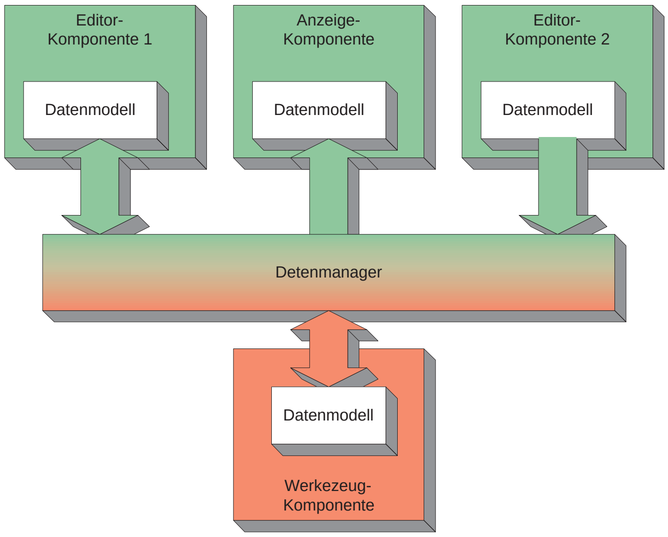


Abbildung 5.9.: SDMV: Der Datenmanager

benötigt also eine zentrale Komponente, die den Austausch von Änderungsnachrichten zwischen den einzelnen Datenmodellen organisiert. Diese Komponente heißt *Datamanager*.

Ein weiterer Punkt ist ebenfalls noch einzukalkulieren. Die einzelnen Editoren und Viewer sollten austauschbar sein. Durch einen Zugriff über entsprechend entworfene Schnittstellen ist das auch kein Problem, allerdings sollte man nicht so vermessen sein, zu glauben, bereits jetzt vorauszusehen, welche Anforderungen zukünftige Werkzeuge beispielsweise an einen Diagrammeditor stellen werden und welche Überraschungen zukünftige Versionen von Java bereithalten. Der Grundstein für die leichte Wartbarkeit eines Systems wird bereits beim Entwurf gelegt. Für den Fall, daß sich der Zugriff auf einen Editor doch eines Tages ändert, sollte man nicht den gesamten Programmcode durchforsten müssen um einige Kleinigkeiten zu ändern.
Vielmehr liegt es nahe, schon im Sinne einer besseren Strukturierung des Gesamtsystems, die für den Abgleich der Teildatenbasen der Editoren und Anzeigen notwendigen Konvertierungen in editorspezifische Module zu verlegen, die quasi einen *Adapter* für den Anschluß eines Editors darstellen. Sollte sich dann eines Tages die Anschlußnorm ändern, dann kann der von Änderungen betroffene Code leicht identifiziert und alle Änderungen an einem Platz durchgeführt werden.

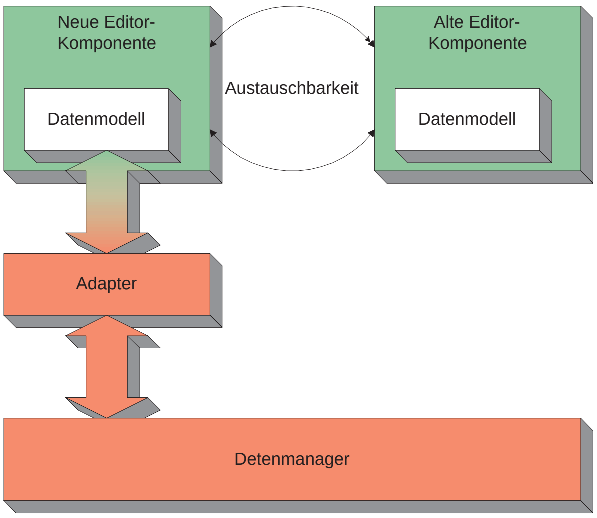


Abbildung 5.10.: SDMV: Austauschbarkeit der Editorkomponenten

Zum Schluß ist noch ein Mechanismus vorgesehen, der es Programmodulen, die auf dem Gesamtdatenmodell arbeiten, sogenannten *Akteuren*, erlaubt, einer Änderung des Datenzustandes zu widersprechen, die über eine Angleichoperation von einem der Editoren erzeugt wurde. Die Änderung darf dann gar nicht erst durchgeführt werden, oder muß rückgängig gemacht werden.
Das endgültige Konzept sieht nun folgendermaßen aus: Eine zentrale Komponente, der *Datamanager* organisiert die Kommunikation der unterschiedlichen Datenmodelle, konkret der vielen Teildatenbasen mit einer Gesamtdatenbasis. Die Gesamtdatenbasis ist direkt an den *Datamanager* angeschlossen, die Teildatenbasen über datenspezifische *Adapter*, die sowohl eine Konvertierung der Daten,

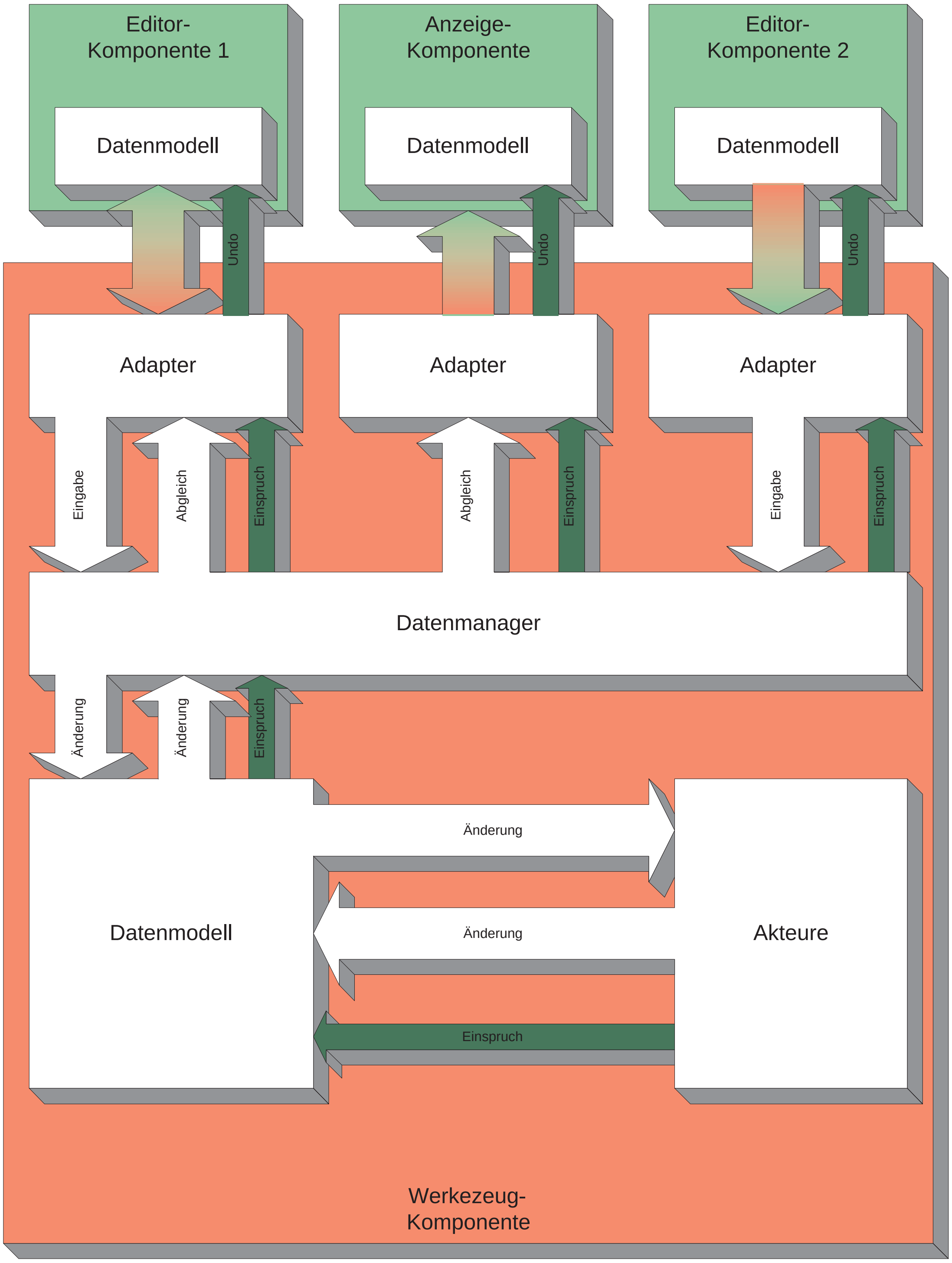


Abbildung 5.11.: SDMV: Übersicht

als auch eine Konvertierung der Nachrichten in eine einheitliche, zur Gesamtdatenbasis passende Form bringen.
Eine Änderung in der Gesamtdatenbasis wird an den *Datamanager* weitergegeben. Dieser verteilt die Änderungsnachricht an die *Adapter* und diese leiten sie in konvertierter Form an die *Views*. Eine Änderung in einem der Editoren wird über den jeweiligen *Adapter* des Editors an den *Datamanager* gemeldet. Dieser leitet die Nachricht direkt an die Gesamtdatenbasis weiter. Von dort wird die Änderung allen Akteuren bekannt gemacht, die auf der Gesamtdatenbasis arbeiten. Erfolgt ein Einspruch, wird die Änderung rückgängig gemacht, oder garnicht erst durchgeführt. Bei erfolgreicher Änderung werden auch die restlichen *Views* von der Änderung in Kenntnis gesetzt. Dies geschieht wiederum unter Zuhilfenahme der jeweiligen *Adapter*.

5.1.4. Parsen graphischer Information

Die Anforderungen und die Architektur machen es notwendig, daß ein in graphischer Form vorliegendes Automatendiagramm in eine abstrakte Datenstruktur geparst werden muß.
Aufgrund der Tatsache, daß zur Eingabe des Diagramms ein semantikerfreier Editor benutzt wird, ist nicht möglich, die Diagrammsymbole direkt mit einer festen Bedeutung zu versehen. Der Editor kennt nämlich keine Zustände und Transitionen, sondern nur Rechtecke, Text, Linien und Pfeile. Würde für die Anwendung des Zeichnens eines Automatendiagramms ein spezialisierter Editor programmiert, könnte eine direkte Zuordnung zwischen graphischer Darstellung und Datenmodell des Automaten hergestellt werden. Mit der Erzeugung eines neuen Symbols, würde automatisch ein entsprechendes Objekt im Datenmodell erzeugt. Die gesamte weitere Bearbeitung würde auf Basis des Datenmodells des Automaten stattfinden. Zwei Zustände würden nicht mit einer Linie verbunden, sondern es würde eine neue Transition erzeugt, die Quell- und Zielzustand miteinander verbindet.
Obwohl dieses Verfahren zunächst einfacher und besser erscheint, hat es bestimmte Nachteile. Zunächst ist der programmierte Editor nur für diesen einen Zweck einzusetzen. Ein Objektmodelldiagramm beispielsweise erfordert dann einen eigenen Editor, der Klassen und Vererbungsbeziehungen ‘kennt’ und ‘weiß’, wie diese darzustellen und auch zu manipulieren sind. Die Zahl der zu wartenden Editoren steigt also mit der Anzahl der verwendeten Diagramme. Zum anderen erlegt ein solcher Editor dem Benutzer in manchen Fällen unnötige Beschränkungen auf. Das OEF benutzt ein Dokument zur Eingabe von Informationen mit dem Hintergedanken, daß Benutzer über eine Unterstützung von notwendigen Dokumentationsarbeiten schrittweise an die Verwendung von Entwurfswerkzeugen herangeführt werden können. Sämtliche Verarbeitungsfunktionen müssen also deaktivierbar sein. Bei einem Editor der zuerst beschriebenen Art ist dies aber nicht möglich, da er ja nicht Linien und Rechtecke verarbeitet, sondern Zustände

und Transitionen eines Modells, das eine bestimmte Syntax benötigt. Selbst wenn also alle Funktionen abgeschalten sind, kann sich ein Benutzer durch die vorgegebene Syntax behindert fühlen und das, obwohl er nicht vor hatte, irgendeine erweiterte Funktionalität zu benutzen.
Für Werkzeuge der Plattform OEF wurde deshalb entschieden, Eingabe und Verarbeitung zu trennen. Dadurch ergibt sich aber das Problem, eine zunächst lose Ansammlung von graphischen Informationen in eine abstrakte Datenstruktur zu überführen, die einem gewünschten Metamodell entspricht. Die Problematik ist verwandt mit der Aufgabe, eine zunächst lose Ansammlung von Buchstaben aus einem Texteditor in einen abstrakten Syntaxbaum zu parsen um die so gewonnene Eingabesprache dann beispielsweise in eine andere zu compilieren.
Zur Lösung könnte man wie bei einem textbasierten Parser von einer geeignet strukturierten Menge atomarer Eingabesymbole ausgehen und diese in Schlüsselsymbole gruppieren. Von der Art der Strukturierung hängt es im wesentlichen ab, wie leicht oder schwierig die Auflösung und Analyse derselben ist.
Je mehr Strukturierung man bereits innerhalb des Editors vorgibt, desto leichter wird die Analyse, aber desto weniger allgemein verwendbar wird der Editor. Die geringste Form der Strukturierung ist sicher, dem Benutzer völlig freie Hand bei der Eingabe zu lassen und vor der Analyse zusammenhängende oder überlappende Grafikelemente zu gruppieren. Ein Rechteck, das von einem Text überlappt wird, könnte also einen Zustand darstellen. Je größer die Zahl von Schlüsselsymbolen und je größer die Ähnlichkeit zwischen diesen ist, desto schwieriger wird die Auflösung. Zudem ist es beispielsweise nicht möglich, den Text des Zustandes nicht innerhalb des Rechtecks zu positionieren, sondern außerhalb davon.
Der nächste logische Schritt wäre, dem Benutzer die Gruppierung zusammengehörender Elemente zu überlassen, beispielsweise mit einer aus vielen Grafikprogrammen bekannten Funktion, die es erlaubt, mehrere selektierte Elemente fortan als ein einziges Element zu behandeln. Das Problem, den Text des Zustandes frei zu positionieren wäre damit gelöst, ebenso Auflösungsprobleme bei ähnlichen Schlüsselsymbolen. Die Kosten dieses Verfahrens entstehen hier jedoch dem Benutzer, der viel Zeit damit verliert, für ihn unnötige Arbeiten zu erledigen und dabei noch jede Menge Fehler begehen kann und dann korrigieren muß, da man schnell einmal vergessen kann, zwei Elemente zu gruppieren, zumal wenn die Gruppierung im Diagramm unsichtbar ist, was sie aber sein sollte, da Diagramme auch ohne sichtbare Gruppierungsinformationen bereits mit Überfrachtung zu kämpfen haben.
Bei den ersten beiden Verfahren hat der Benutzer zwar viel Freiheit, aber diese macht ja eigentlich nur Sinn, wenn er die erweiterte Funktionalität nicht benutzen will. Andernfalls muß er sich sowieso an eine vorgegebene Notation halten, weswegen er mit Recht auf eine bequeme und zeitsparende Eingabe wertlegen wird. Wiederkehrende Schlüsselsymbole sollten vorgefertigt sein, mit anderen Worten, der Benutzer fordert einen Editor, der sich für ihn selbst nicht von einem Editor unterscheidet, der speziell für diesen Anwendungszweck programmiert wurde.

Als Lösung bietet sich an, den Editor so zu gestalten, daß es möglich ist, vorgruppierte Symbole zu verwenden, die für den Benutzer bereits mit einer bestimmten Semantik verbunden sind, die sich beispielsweise aus einer Bezeichnung, gepaart mit dem momentanen Verwendungszweck ergibt. Für den Editor selbst wäre es nach wie vor nur eine Ansammlung von grafischen Symbolen ohne übergeordnete Semantik.
Damit wäre das Strukturierungsproblem bereits mit einem minimalen Aufwand für die Analyse bei maximaler Wiederverwendbarkeit des Editors gelöst. Für die Analyse bleibt dann nur noch die Identifizierung der vorstrukturierten Schlüsselsymbole, man müßte also nur ein Schlüsselsymbol untersuchen, ob es aus einem Rechteck und einem Text besteht und könnte es dann zugleich als Zustand identifizieren.
Mit einem kleinen Trick kann man sich aber auch diese Arbeit stark vereinfachen, was neben höherer Performance noch den Vorteil besitzt, daß eine Modifikation der graphischen Darstellung leichter möglich ist.
Im vorigen Fall müßte man, wenn man Zustände aus unerfindlichen Gründen plötzlich als Ellipsen mit Text darstellen wollte, einerseits die vordefinierten Symbole für den Editor ändern und andererseits auch noch das Analysemodul dahingehend modifizieren, daß nun ein Schlüsselsymbol, das aus einer Ellipse und Text besteht, als Zustand identifiziert wird, bzw. man müßte schon vorher ein Analysemodul programmiert haben, das Schlüsselsymbole über einen Vergleich mit gespeicherten Vorlagensymbolen identifiziert.
Aber es geht auch einfacher. Man versieht die vorgruppierten graphischen Elemente mit einem Verweis auf die Vorlage, aus der sie erzeugt wurden. Der damit erzielte Gewinn rechtfertigt den zusätzlichen Speicherplatzverbrauch und man kann auf diese Weise die Schlüsselsymbole identifizieren, ohne auf ihre grafische Beschaffenheit eingehen zu müssen. Eine Elementgruppe ist ganz einfach dann ein Zustand, wenn sie aus einer Vorlage mit der Bezeichnung 'Zustand' erzeugt wurde.
Auf diese Weise kann später die grafische Darstellung geändert werden, ohne daß weitere Modifikationen am Analysemodul notwendig werden.
Bleibt noch ein letztes Problem, nämlich auf effektive Weise syntaktische Verbindungen zwischen Schlüsselsymbolen zu bestimmen, beispielsweise mit welchen Zuständen eine Transition verbunden ist.
Auch hier könnte man versucht sein, auf niedrigem Level zu bestimmen, welche Rechtecke eine Linie verbindet und daraus Rückschlüsse zu ziehen, welcher Zustand nun gemeint ist. Dies auf rein graphischem Weg zu bestimmen ist wiederum sehr aufwendig und die zugehörige Auswertung sehr zeitintensiv.
Das Problem ist aber sehr ähnlich dem ersten. Auch hier erzielt man die optimale Lösung dadurch, daß man so viel Arbeit wie möglich aus dem Analysemodul in den Editor verlagert, ohne daß die Wiederverwendbarkeit darunter leidet.
Indem man den Editor um das Konzept von Klebepunkten (siehe [6]) erweitert erhält der Benutzer einen erhöhten Bedienkomfort und der Programmierer eine

komfortable Möglichkeit, syntaktische Verbindungen aufzuspüren.
Klebepunkte sind nichts anderes als vordefinierte Einraststellen für Verbindungslinien. Der Benutzer hat den Vorteil, daß die Verbindungslinien wie Gummibänder mitgeführt werden, falls das verbundene Element bewegt wird. Gerade bei der Konstruktion von Diagrammen kommt es häufig vor, daß Symbole anders positioniert werden, um ein übersichtliches und aussagekräftiges Gesamtbild zu erzeugen. Ohne Klebepunkte müßten die Verbindungen jedesmal von Hand angepaßt werden, falls es zu einer Neupositionierung eines Symbols kommt. Diese Klebepunkte müssen natürlich im Datenmodell des Editors als Verweis von einem grafischen Symbol (z.B. Rechteck) auf ein anderes (z.B. ein Linienelement) berücksichtigt werden. Für die Analyse bedeutet dieser Umstand, daß auch syntaktische Verbindungen bereits implizit in der Datenstruktur des Editors gespeichert werden.
Folglich ist es also möglich den eigentlich semantikfreien Editor bereits indirekt mit den wichtigsten Informationen für die syntaktische Analyse anzureichern, ohne daß Wiederverwendbarkeit oder Benutzerfreundlichkeit darunter leiden.
Bei der vielzitierten Ähnlichkeit mit klassischem Parsen von Text stellt sich natürlich fast automatisch die Frage, ob man vielleicht in der Lage wäre, Werkzeuge mit der Funktionalität von Lexx und Yacc für graphische Notationen anzugeben, bzw. wie eine adäquate Grammatik für eine graphische Notation angegeben werden könnte.
Unter Umständen genügt es, die für Textparser übliche Form etwas zu erweitern. Ein Versuch für den konkreten Fall der Automatendiagramme könnte aussehen wie Beispiel 5.1. Zusätzlich müssen Schlüsselsymbole an geeigneter Stelle in Formen (shapes) und Verbindungen (connections) unterschieden werden.

topLevel	::=	$\{$transition$\}^*$
transition	::=	state **TRANSITION** \traid\ state
	|	**TRANSITION** \traid\ state
state	::=	**STATE** \staid\

Tabelle 5.1.: Graphische Notation für Automatendiagramme

Als Einschränkung wird vereinbart, daß ein graphisches Schlüsselsymbol maximal drei Textfelder besitzen kann. Eine Form besitzt maximal ein Textfeld, eine Verbindung maximal drei. Es ist zu beachten, daß die Textfelder nicht Teil der graphischen Grammatik sind (da es sich ja um keine Grafik handelt), sondern separat wie Text geparst werden und dafür eine eigene Grammatik anzugeben ist. Wie diese Textfelder geparst werden können, wird im nächsten Abschnitt behandelt. Textfelder werden in der graphischen Grammatik als Parameter der Form \<parameter>\ angegeben.

Wie man an Beispiel 5.1 sehen kann, ist die graphische Notation für Automatendiagramme relativ einfach, so daß auf die Entwicklung lexx/yacc-ähnlicher Werkzeuge an dieser Stelle verzichtet werden kann, was im übrigen auch den Rahmen dieser Arbeit sprengen würde. Die Thematik ist aber interessant genug, sie an anderer Stelle weiterzuverfolgen.
Da ein Großteil der Analyse bereits im Editor erledigt wird, ist es nicht mehr schwer, einen Parser für Automatendiagramme von Hand zu programmieren. Aufgrund der Einfachheit ist es sogar möglich, den Vorgang bidirektional zu gestalten, d.h. eine Diagrammgrafik nicht nur als Eingabe, sondern auch als Ausgabe zu verwenden, die Änderungen an der abstrakten Datenstruktur wieder in eine Diagrammgrafik umsetzt.
Die nötigen Routinen werden in die SDMV-Architektur (siehe 5.1.3) eingepaßt und wie dort vorgeschlagen in einem eigenen Modul (Adapter) zusammengefaßt.

5.1.5. NLTF-Parsen

Das NLTF-Parsen ist die Bezeichnung für das Problem, einzelne Codefragmente ohne lineare Ordnung, wie sie beispielsweise in Automatendiagrammen vorkommen können, in eine abstrakte Datenstruktur zu überführen. Ob es sich dabei um eine gemeinsame Datenstruktur für graphische und textuelle Informationen handelt, oder ob der Textanteil einer nicht rein graphischen Notation separat verwaltet wird, ist dabei unerheblich.
Tatsache ist, daß in den Textfeldern in der Regel eine Sprache verwendet wird, die sonst zeilenorientiert in Form eines Quelltextes zum Einsatz kommt. Leider sind fast alle vorhandenen Parser oder Parserwerkzeuge auf diesen Verwendungszweck hin optimiert. Ein zeilennummerorientiertes Fehlermanagement ist nur ein Beispiel.
Das größte Problem jedoch ist, daß zeilenorientierte Parser einen festen Kontext vorgeben, unter dem jedes Codefragment betrachtet wird. Die meisten Parser kennen zwar mehrere Ebenen, so daß es möglich ist, von einer Ebene Beziehungen zu darüberliegenden Ebenen herzustellen, jedoch verbietet es das Konzept der Ebene, bestimmte Bereiche der selben, oder darüberliegender Ebenen aus- und andere explizit einzuschließen.
Für Diagramme und insbesondere für Automatendiagramme ist das Konzept der Kontextebenen jedoch völlig unzureichend. Benötigt wird vielmehr ein Konzept von atomaren Blasen, die in einer frei definierbaren Kontextmatrix miteinander in Beziehung gesetzt werden können.
So steht das Vorbedingungsfeld einer Transition im Kontext mit genau einem Zustand und dem Eingabepattern, das Nachbedingungsfeld im Kontext mit allen Feldern der Transition, dem Quellzustand und dem Zielzustand, aber nicht zu anderen Transitionen und Zuständen.
Ein zeilenorientierter Parser läßt sich zwar zu einem gewissen Prozentsatz an die genannten Erfordernisse anpassen, jedoch wird man damit niemals die volle

Leistungsfähigkeit erreichen.

Ein echter NLTF-Parser könnte auf zweierlei Weise mit konventionellen Mitteln erzeugt werden. Im einen Fall wird die Kontextmatrix fest im Parser verdrahtet, indem zunächst eine neue Sprache konstruiert wird, die äquivalent zur in einem bestimmten Diagrammtyp verwendeten graphischen und textuellen Information ist, d.h. alle in einem Diagramm enthaltene Information graphischer oder textueller Natur kann mit dieser Sprache in rein textueller Form formuliert werden.

Der Vorteil dieses Verfahrens ist, daß man auf bewährte Mittel und Wege der Parsererstellung zurückgreifen kann. Nachteile ergeben sich vor allem dadurch, daß man eine effiziente Sprache finden muß, die den gewünschten Diagrammtyp abbildet. Man benötigt weiterhin eine zusätzliche Übersetzungsebene; schließlich wird zunächst die abstrakte Datenstruktur, welche graphische und Textinformation enthält, komplett in einen Text übersetzt, der dann wiederum in eine zweite abstrakte Datenstruktur geparst wird, welche die gleichen Informationen enthält, nur anders aufbereitet.

Die zweite Möglichkeit einer Herangehensweise ist, auf eine Festlegung der Kontextmatrix zu verzichten und statt dessen Konstrukte einzuführen, die es erlauben, ein beliebiges Stück Quelltext zu kapseln und mit einem Identifikator sowie einer Liste von Kontextbeziehungen zu versehen.

Dies hätte den Vorteil, daß ein solcher Parser prinzipiell für mehrere Diagrammtypen, welche die gleiche Sprache zur Beschriftung verwenden, geeignet ist. Die Entwicklung eines solchen Parsers ist jedoch komplizierter, hätte aber den Vorteil, daß man mit einer einzigen abstrakten Datenstruktur auskommen kann. Es muß auch nicht das komplette Diagramm in eine Eingabesprache übersetzt werden, sondern nur die Kontextbeziehungen des Diagramms in einer für den Parser lesbaren Form ausgegeben werden. Die Ergebnisse des Zerteilvorgangs könnten über den Identifikator leicht dem Quelltextfragment zugeordnet werden, so daß einerseits die gleiche abstrakte Datenstruktur zur Speicherung der Resultate verwendet werden könnte, andererseits ein adäquater Mechanismus zur Fehlerzuordnung ohne Zeilennummern zur Verfügung steht.

Ein Vorteil den alle NLTF-Parser besitzen ist, daß durch inkrementelles Parsen eine hohe Änderungseffizienz erreicht werden kann. Ändert sich ein Feld des Diagramms, muß nicht komplett neu geparst werden, sondern es reicht, das geänderte Feld an den Parser zu übergeben. Dieser kann, ausgehend vom letzten Parserlauf nur dieses eine Feld parsen und den zugehörigen stark eingeschränkten Kontext prüfen.

Wegen der stärkeren feldorientierten Dateneingabe und -ausgabe dürfte ein nach dem zweiten Verfahren erstellter Parser noch besser für die Realisierung des inkrementellen Parsens geeignet sein, als Parser der ersten Art.

5.2. Die Kernkomponente ‘STD-Assistant’

5.2.1. Architektur

5.2.1.1. Zerlegung in Teilsysteme

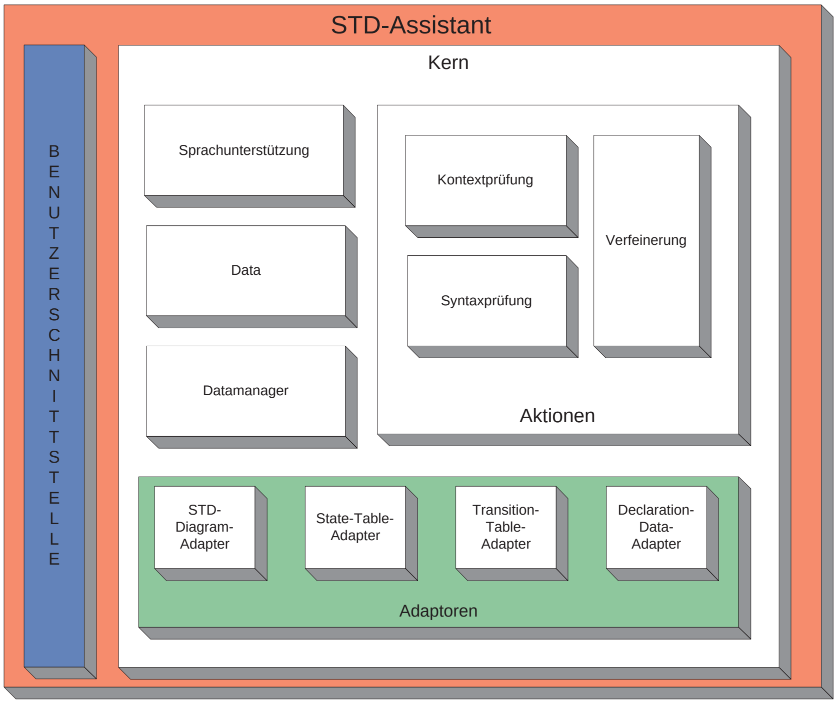


Abbildung 5.12.: Architektur

Die Zerlegung der Komponente in einzelne Subsysteme ergibt sich im wesentlichen aus den in den Anforderungen vorgegebenen Aufgaben und der Einbeziehung der SDMV-Architektur.

Die Kernkomponente zerfällt zunächst in den Kern und die Benutzerschnittstelle. Der Kern besteht aus dem SDMV-System samt angeschlossenem Datenmodell sowie einem Pool von *Akteuren* und der als eigenes Modul realisierten Sprachunterstützung. Ein *Akteur* ist ein Programmodul, das eine der drei Hauptfunktionalitäten der Kernkomponente implementiert (Syntax, Kontext, Verfeinerung). Es gibt zwei Standardakteure (Syntax und Kontext) und optionale Akteure. Momentan gibt es nur einen optionalen Akteur, die Verfeinerung. Module für weitere Akteure könnten jederzeit hinzugefügt werden.

5.2.1.2. Aufgaben der Teilsysteme

Datamanager
Der Datamanager ist die in Abschnitt 5.1.3.2 beschriebene Zentrale des Informationsabgleichs zwischen den Views/Editoren und dem Gesamtdatenmodell.

Adapter
Die Adapter besorgen den in Abschnitt 5.1.3.2 beschriebenen Anschluß von austauschbaren Komponenten zur Anzeige und Eingabe von Information. Es gibt im ganzen vier Adapter. Drei davon dienen zum bidirektionalen Anschluß der Editoren für das Automatendiagramm, die Zustandstabelle und die Transitionstabelle. Ein vierter Adapter erledigt einen unidirektionalen Anschluß eines Texteditors, mit dem eine externe Informationsquelle zur Eingabe von Daten simuliert wird, die nicht direkt zum Automatendokument gehören, oder nicht selbständig vom Datenmodell aus geändert werden können. Dazu gehört:

- Die Eingabemethoden der Klasse.
- Die Ausgabemethoden der Klasse.
- Die der Klasse zugänglichen Referenzen.
- Die Attribute der Klasse.

Datenmodell
Das Datenmodell ist eine abstrakte Umsetzung des in 2.3 beschriebenen Automatendokumentes. Zusätzlich verwaltet das Datenmodell einen Verweis auf die im jeweiligen Automatendokument verwendete Sprache, die ja beliebig sein kann, sofern sie die notwendigen prädikatenlogischen und funktionalen Ausdrücke enthält sowie evt. Ergebnisse von Syntax- oder Kontextprüfung.

Syntaxprüfung
Die Syntaxprüfung ist ein Standardakteurmodul. Es realisiert die in Abschnitt 4.3 spezifizierte Aufgabe der Prüfung auf syntaktische Korrektheit des im Datenmodell gespeicherten Automatendokumentes.

Kontextprüfung
Die Syntaxprüfung ist ein Standardakteurmodul. Es realisiert die in Abschnitt 4.3 spezifizierte Aufgabe der Prüfung auf semantische Korrektheit des im Datenmodell gespeicherten Automatendokumentes.

Verfeinerung
Die Verfeinerung ist ein optionales Akteurmodul. Es realisiert die in Abschnitt 2.4 spezifizierte Aufgabe der Umsetzung eines Verfeinerungsverfahrens für Automatendokumente. Die Verfeinerung besitzt eine eigene Benutzerschnittstelle, die an die Hauptkomponente durchgereicht wird.

Sprachunterstützung
Die Sprachunterstützung enthält nicht die gesamte Sprache, sondern nur ein Interface, das einen sprachunabhängigen Zugriff ermöglicht sowie die notwendigen Anpassungsarbeiten, die um so geringer ausfallen, je spezieller die verwendete Sprache auf die Verwendung in Automatendokumenten abgestimmt ist. Die Sprache selbst (Parser etc.) ist ein eigenständiges Hilfswerkzeug und wird über das OEF angesprochen.

Benutzerschnittstelle
Die Benutzerschnittstelle umfaßt sämtliche Funktionen, welche die Interaktion zwischen Benutzer und der Kernkomponente betreffen.
Die Benutzerschnittstelle erledigt folgende Unteraufgaben:

- Darstellung und Auswertung eines Kontroll- und Steuerpaneels.
- Verwaltung eines Satzes von Regeln, mit denen festgelegt werden kann, welche Funktionen der Benutzerschnittstelle momentan zugänglich sind.
- Übergabe von Steuerelementen an die Werkzeugplattform OEF, d.h. Konfiguration der werkzeugübergreifenden Benutzerschnittstelle des OEF, beispielsweise durch Erweiterung der Werkzeugleiste des OEF.
- Annahme von Erweiterungen der Benutzerschnittstelle, wie sie von optionalen Akteuren angefordert werden können, beispielsweise Annahme weiterer Toolbuttons zur Steuerung der Verfeinerung.

Es ist zu beachten, daß jede Komponente des Gesamtsystems, die zugleich eine eigenständige OEF-Komponente darstellt, ihre eigene Benutzerschnittstelle besitzt. Diese sind eigenständig, werden aber von der Benutzerschnittstelle des Kernsystems an die jeweiligen Bedürfnisse angepaßt, d.h. konfiguriert, falls das möglich und notwendig sein sollte.

5.2.2. Kommunikationsfluß

5.2.2.1. Datenmodell $\longleftrightarrow$ Datamanager $\longleftrightarrow$ Adapter

Die Kommunikation zwischen dem Datamanager und den Adapter der Editor- und Anzeigekomponenten entspricht im wesentlichen dem in 5.1.3.2 beschriebenen. Eine relevante Änderung im Datenmodell der Editoren, die auch Auswirkungen auf die Datenbasis des Kerns hat, wird vom Adapter an den Datamanager gemeldet. Dieser meldet die Änderung an das Datenmodell des Kerns und wartet, bis er von diesem eine Bestätigung, oder eine Ablehnung erhalten hat. Im Falle einer Ablehnung, teilt er diese dem Adapter mit. Im Falle einer Bestätigung, sorgt der Datamanager dafür, daß auch die anderen Adapter von der Änderung unterrichtet werden, damit auch sie ihre Darstellung anpassen können. Eine Änderung der Datenbasis des Kerns wird an den Datamanager gemeldet, der diese dann an alle Adapter verteilt. Diese entscheiden dann selbständig, ob die Änderung sie betrifft und passen sich, falls dem so ist, an die Änderung an.

5.2.2.2. Syntaxprüfung ⟷ Sprachunterstützung

Die Syntaxprüfung übergibt ein Sprachfeld, beispielsweise ein Zustandsprädikat an die Sprachunterstützung und holt sich im Anschluß daran ein abstraktes Sprachelement und eine Liste mit Fehlern ab.

5.2.2.3. Kontextprüfung ⟷ Sprachunterstützung

Die Kontextprüfung übergibt ein Sprachfeld des Datenmodells und den zugehörigen Kontext. Die Übergabe erfolgt als Gesamtelement, so daß die Sprachunterstützung auf die Ergebnisse der Syntaxprüfung in Form der abstrakten Sprachelemente zurückgreifen kann. Nach erfolgter Prüfung erhält die Kontextprüfung das Ergebnis durch ein eventuell modifiziertes abstraktes Sprachelement und eine Liste von Fehlern zurück.

5.2.2.4. Verfeinerung ⟷ Sprachunterstützung

Die Verfeinerung übergibt an die Sprachunterstützung die Art einer Beweisverpflichtung und die dafür benötigten Daten. Als Ergebnis erhält die Verfeinerung die Beweisverpflichtungen zurück.

5.2.2.5. Verfeinerung ⟷ Syntaxprüfung

Die Verfeinerung bedient sich der Syntaxprüfung um sicherzustellen, daß:

- das Automatendokument syntaktisch korrekt ist, bevor die Verfeinerung aktiviert wird.
- im Laufe eines Verfeinerungsschrittes eingegebene Informationen vor Abschluß des Schrittes syntaktisch korrekt sind.

5.2.2.6. Verfeinerung ⟷ Kontextprüfung

Die Verfeinerung bedient sich der Kontextprüfung um sicherzustellen, daß:

- das Automatendokument einen korrekten Kontext besitzt, bevor die Verfeinerung aktiviert wird.
- im Laufe eines Verfeinerungsschrittes eingegebene Informationen vor Abschluß des Schrittes in korrektem Kontext zum bestehenden Dokument steht.

5.2.2.7. Verfeinerung $\longleftrightarrow$ Benutzerschnittstelle

Bei Aktivierung sorgt die Verfeinerung dafür, daß die Benutzerschnittstelle so modifiziert wird, daß nur noch fest definierte Konstruktionsschritte ausgeführt werden können. Dazu werden die Regeln der Benutzerschnittstelle modifiziert sowie ein neues Schnittstellenmodul übergeben, das die neuen Funktionen steuert.

5.2.2.8. Datenmodell $\longleftrightarrow$ Akteur

Das Datenmodell unterrichtet die Akteure von Änderungen der Datenbasis. Dies wird benutzt, um

- die Prüfungen von Syntax und Kontext durchzuführen, die schnell erledigt werden können, um dem Benutzer auf eine Eingabe sofort erste Hinweise auf mögliche Fehler geben zu können, ohne den Programmablauf zu verzögern.
- Akteuren eine Einspruchsmöglichkeit zu geben. So sind bei aktivierter Verfeinerung abhängig vom jeweiligen Zustand bestimmte Änderungen nicht erlaubt. Oft ist das stark kontextabhängig, so daß nicht schon in den semantikfreien Editoren darüber entschieden werden kann, ob eine Aktion zulässig ist.

5.2.3. Datenstrukturen - Beschreibung der Klassen

- Paket `default`

  Dieses Paket enthält den Part `STD Assistant` mitsamt aller seiner Unterkomponenten.

  - `STDAssistant`
    Ist die Hauptkomponente.
  - `STDAssistantCore`
    Kern und Datenhaltung.
  - `STDAssistantUI`
    Benutzerschnittstelle.
  - `STDAssistantSI`
    Interface für OEF Skriptverarbeitung.
  - `STDAssistantSIAdapter`
    Implementierung der OEF Skriptverarbeitung.

- Paket `action`

  Dieses Paket enthält alle Aktionen (Akteure) wie Syntax- und Kontextprüfung (siehe auch Abschnitt 5.1.3).

  - `Action_ContextCheck`
    Kontextprüfung.
  - `Action_SyntaxCheck`
    Syntaxprüfung.
  - `Action_Refinement`
    Verfeinerung.
  - `Action_RefinementSI`
    Interface für OEF Skriptverarbeitung.
  - `Action_RefinementUI`
    Benutzerschnittstelle der Verfeinerung.
  - `ActionRules`
    Ein Satz von Konfigurationsregeln. Siehe auch Abschnitt 5.2.6.7)
  - `FriscoAction`
    Superklasse aller Aktionen.
  - `RefinementUIAction`
    Superklasse aller Swing-Actions für die Benutzerschnittstelle der Verfeinerung. Nicht zu verwechseln mit Aktionen (Akteuren)

- Paket `configrules`

  Dieses Paket enthält alle Schnittstelle und Klassen für eine Laufzeitkonfiguration von Benutzerschnittstellen durch fremde Komponenten. Die einzelnen Klassen sind Abschnitt 5.2.6.7 beschrieben.

- Paket `connections`

  Dieses Paket enthält alle Verbindungselemente zu den anderen Parts, die in einem Automatendokument benötigt werden.

  - `FDEHookUpConnection`
    Verbindung zu einem Diagrammpart.
  - `ProofConnection`
    Verbindung zu einem Beweismanager
  - `TableConnection`
    Verbindung zu einem Tabellenpart.
  - `TextConnection`
    Verbindung zu einem Textpart.

- Paket `datamanager`

  Dieses Paket enthält das interne Datenmodell und die Implementierung des SDMV-Mechanismus.

- Datenmodell
  - ▷ `AbstractDocumentStringObject`
    Superklasse aller Datenelemente mit Text.
  - ▷ `AbstractDocumentContainer`
    Behälter für mehrere Datenelemente.
  - ▷ `AbstractSTDDocumentObject`
    Superklasse aller Datenelemente.
  - ▷ `AttributeDeclaration`
    Datenelement.
  - ▷ `AttributeDeclarations`
    Datenelement.
  - ▷ `AutomatonDocument`
    Zusammenfassung aller Datenelemente in einer Datenstruktur.
  - ▷ `AutomatonDocumentSI`
    Skriptinterface der Datenstruktur.
  - ▷ `InMethodDeclaration`
    Datenelement (gehört zu den Deklarationen).
  - ▷ `InMethodDeclarations`
    Datenelement (gehört zu den Deklarationen).
  - ▷ `ObjectDeclaration`
    Datenelement (gehört zu den Deklarationen).
  - ▷ `ObjectDeclarations`
    Datenelement (gehört zu den Deklarationen).
  - ▷ `OutMethodDeclaration`
    Datenelement (gehört zu den Deklarationen).
  - ▷ `OutMethodDeclarations`
    Datenelement (gehört zu den Deklarationen).
  - ▷ `State`
    Datenelement.
  - ▷ `StateCondition`
    Datenelement (gehört zu einem Zustand).
  - ▷ `StatePattern`
    Datenelement (gehört zu einem Zustand).
  - ▷ `Transition`
    Datenelement.
  - ▷ `TransitionInputPattern`
    Datenelement (gehört zu einer Transition).
  - ▷ `TransitionOutput`
    Datenelement (gehört zu einer Transition).

  - ▷ `TransitionPostcondition`
    Datenelement (gehört zu einer Transition).
  - ▷ `TransitionPrecondition`
    Datenelement (gehört zu einer Transition).
  - ▷ `UnknownElement`
    Unbekanntes Datenelement.
- SDMV-Implementierung
  - ▷ `AbstractSTDEvent`
    Nachricht von Datenstruktur an Datamanager.
  - ▷ `AbstractSTDEventListener`
    Nachrichtenempfänger.
  - ▷ `DataManager`
    Zentrale für Datensynchronisation.
  - ▷ `DeclData_Access`
    Adapter für das Deklarationstextfeld.
  - ▷ `ErrorItem`
    Fehlermeldung.
  - ▷ `ExtSTDSIAdapter`
    Filter für vertikale Kommunikation des Diagrammparts.
  - ▷ `ExtTBLSIAdapter`
    Filter für vertikale Kommunikation des Tabellenparts.
  - ▷ `SimpleData_Access`
    Superklasse aller Adapter.
  - ▷ `STDGraphicData_Access`
    Adapter für das Diagramm.
  - ▷ `STDSIAdapterEvent`
    Nachricht von `ExtSTDSIAdapter`.
  - ▷ `STDSIAdapterEventListener`
    Nachrichtenempfänger.
  - ▷ `TableAccess`
    Superklasse aller Tabellenadapter.
  - ▷ `TBLSIAdapterEvent`
    Nachricht von `ExtTBLSIAdapter`.
  - ▷ `TBLSIAdapterEventListener`
    Nachrichtenempfänger.
  - ▷ `TBLStateData_Access`
    Adapter für die Zustandstabelle.
  - ▷ `TBLTransitionData_Access`
    Adapter für die Transitionstabelle.

- Paket `languagesupport`

  Dieses Paket enthält die Interfaces der Sprachunterstützung. Eine Beschreibung findet sich in Abschnitt 5.2.6.6.

- Paket `languagesupport_ff`

  Dieses Paket enthält die Implementierung der Sprachunterstützung für die Sprache FriscoF. Eine Beschreibung findet sich in Abschnitt 5.4.4.

5.2.4. Benutzerschnittstelle

Die Benutzerschnittstelle der Kernkomponente ist in aller Deutlichkeit in Abschnitt 3.1.1 beschrieben.
Die blinkende Fehleranzeige ist ein einfaches `JPanel`, das über einen eigenen Thread gesteuert die Farbe wechseln kann. Die zugehörige Klasse `alertPanel` befindet sich im Paket `gui`

5.2.5. Technische Daten

Archiv:	jar-File
Klasse:	STDAssistant
Klassifizierung:	Documentcontroller
Bezeichnung:	STD Assistant
Version:	1.0
Author(en):	Michael Fahrmair

5.2.5.1. Systemvoraussetzungen

Hardware

CPU:	Pentium ab 200MHz, PPC 604 ab 160MHz, UltraSparc
Hauptspeicher:	ab 32M, empfohlen für das ganze OEF: 64M
Graphikkarte:	schnell, mind. 4M
Monitor:	mind. 17 Zoll
Festplattenplatz:	200k, das ganze OEF inkl. Dokumentation: mind. 50M

Betriebssysteme

Windows95/NT:	inkl. Service-Pack 3 (getestet)
Solaris für Sparc:	(getestet)
Linux:	ab Kernel 2.0.30 (nicht getestet)
MacOS:	ab Version 8.0 (nicht getestet)

Java VM und Klassenbibliotheken

JDK-Version: ab 1.1.5 (bis 1.1.6 getestet)
JFC-Version: ab 1.0.1 (bis 1.0.2 getestet)
OEF-Version: ab 1.2.1

5.2.5.2. Java Packages

JDK: `java.awt`
`java.awt.event`
`java.io`
`java.lang`
`java.math`
`java.text`
`java.util`

Swing: `com.sun.java.swing`
`com.sun.java.swing.border`
`com.sun.java.swing.event`
`com.sun.java.swing.table`
`com.sun.java.swing.text`

OEF: `oef.annotation`
`oef.connection`
`oef.gui`
`oef.help`
`oef.parthandler`
`oef.parthandler.interfaces`
`oef.printing`
`oef.scripting`
`oef.session`
`oef.util`

5.2.5.3. Latex Packages

Keine.

5.2.5.4. Externe Ressourcen

Keine.

5.2.5.5. Unterstützte Interfaces

Aus `oef.parthandler`

Interface	Anmerkung
`PartViewer`	Unterstützt, da vom OEF-System gefordert.
`PartEditor`	ja
`PartWithClipBoard`	-
`PartWithConnections`	-
`PartWithHelp`	ja
`PartWithMenus`	ja
`PartWithPreferences`	-
`PartWithPrinting`	-
`PartWithScaling`	-
`PartWithScriptRecording`	ja
`PartWithScripting`	ja
`PartWithToolBars`	ja
`PartWithUndo`	ja
`PartWithSelection`	-

5.2.6. Implementierungsdetails

In diesem Abschnitt werden die interessantesten implementierungstechnischen Details beschrieben. Aus Gründen des Dokumentationsumfanges und der Aktualität wird jedoch nur ein kleiner Teil behandelt, vornehmlich Details, die nicht auf einfache Weise durch einen Blick in den Quelltext vermittelt werden können.

5.2.6.1. Parthandler-Komponente (`STDAssistant`)

Eine kleine Ungereimtheit des OEF macht während der Initialisierung des Parts einen kleinen Kniff notwendig. Dummerweise werden während der Initialisierung (`initialize` und `initializeFromStream`) angeforderten Verbindungen zu anderen Parthandlern vom OEF erst nach Verlassen der Initialisierungsmethoden initialisiert. Dann liegt die Ablaufkontrolle aber bereits wieder beim Framework. Leider werden aber für die Initialisierung der SDMV-Adapter bereits die Verbindungen benötigt, da ja in den Zielparthandlern teilweise die Scriptinterfaces überschrieben werden müssen.
Aus diesem Grund wird während der Initialisierung ein eigener Thread gestartet, der nur darauf wartet, daß die angeforderten Verbindungen initialisiert werden und darauf dann die bisher verzögerte Initialisierungen vollendet.
Wenn der Fehler des OEF behoben wurde, kann dieses Verfahren entfallen.
Der Kern des Parthandlers (`STDAssistantCore`) nimmt ebenfalls am SDMV-Mechanismus teil, um den Klassennamen und den Typ des Automatendokuments

zu editieren. Es existiert allerdings weder ein Adapter, noch eine Einspruchsmöglichkeit, da die betroffenen Daten nicht synchronisiert werden müssen und auch die Eingabekomponente nicht ausgetauscht werden kann (das hieße ja, ebenfalls den kompletten Datamanager auszutauschen).
Die Lösung sollte aber bei Gelegenheit sauberer und modellkonformer implementiert werden.

5.2.6.2. Der Adapter für die Deklaration (`DeclTextAdapter`)

Dieser Adapter unterscheidet sich von den anderen dreien durch das Fehlen von Synchronisationsroutinen, da der Datenabgleich nur in eine Richtung stattfindet. Die betroffenen Informationen sind nicht vom Gesamtdatenmodell aus änderbar. Der Adapter besorgt den Anschluß an ein `SimpleTextEditor` Objekt. Das größte Problem dabei ist die Auswertung der Änderungsnachrichten. Diese werden bei jedem neuen Buchstaben generiert, der verändert wird. Wird nun bei jedem neuen Buchstaben ein Abgleich der Datenmodelle durchgeführt, ist ein kontinuierliches Schreiben nicht mehr möglich. Aus diesem Grund wird ein eigener Eventpuffer eingerichtet. Dieser läuft im Hintergrund und speichert zunächst alle Events. Nach einem fest definierten Intervall, wird der gesamte Puffer geleert. Dabei werden alle Events, bis auf das letzte, mit einem Flag ausgezeichnet, der darüber informiert, daß das Event nur Teil einer ganzen Folge ist. Der Adapter ignoriert alle Events, bis auf das letzte. So wird sichergestellt, daß auch die kleinste Änderung berücksichtigt wird. Andererseits wird die Eingabe eines längeren Textes erkannt und mit der Angleichung der Datenbasen gewartet, bis die Eingabe beendet, oder eine bestimmte Zeit verstrichen ist.

5.2.6.3. Der Adapter für das Automatendiagramm (`STDGraphicData_Access`)

Die einzig wirklich sehr komplizierte Stelle in diesem Adapter ist die Methode `updateTransitionInDiagram`. Es kann nämlich passieren, daß eine Transition Quell- oder Zielzustand wechselt, worauf dann natürlich die zugehörige Linie neu verbunden werden muß.
Zunächst wird der linke obere Klebepunkt des betroffenen Zustandssymbols identifiziert. Als nächstes wird der Klebepunkt der Transition über den des Zustandes verschoben.
Ein Aufruf der methode `matchGlues()` sorgt dafür, daß übereinanderliegende Klebepunkte verbunden werden.
Ein Aufruf von `connectionCheck()` sorgt nun dafür, daß die Verbindungsstelle von der linken oberen Ecke zu dem Punkt wandert, bei dem eine möglichst kurze Transitionslinie entsteht.
Ein eigener Layoutmechanismus ist noch nicht implementiert. Dies sollte jedoch bald erfolgen, da auf diese Weise natürlich möglicherweise mehrere Transitionen

übereinander gelegt werden, was nicht besonders übersichtlich ist.

5.2.6.4. Der Adapter für die Transitionstabelle (`TBLTransitionData_Access`)

Dieser Adapter leitet sich wie auch der Adapter der Zustandstabelle von der Klasse `TableAccess` ab, hat aber einige Besonderheiten implementiert, die mit der Möglichkeit zusammenhängen, einen Tabelleneintrag für mehrere Transitionen zu benutzen. Auf der anderen Seite muß ein SDMV-Datenobjekt natürlich immer eindeutig einer bestimmten Darstellung zugeordnet werden können. Aus diesem Grund wurde der Adapter für die Transitionstabelle dahingehend modifiziert, daß eine Tabellenzeile mehreren Transitionen zugeordnet werden kann, wenn sich diese ausschließlich in Daten unterscheiden, die nicht in der Transitionstabelle dargestellt werden, d.h. nur Quell- und Zielzustand dürfen sich unterscheiden.
Unterscheiden sich zwei Transitionen dagegen auch in Punkten, die in der Tabelle dargestellt werden, besitzen aber den gleichen Namen, dann werden sie in verschiedenen Zeilen dargestellt.
Dies macht diesen Adapter ein wenig kompliziert, da bei jedem Update von Außen überprüft werden muß, ob nicht eine Transition, die bisher mit ähnlichen zusammengefaßt war, ausgekoppelt werden muß, weil sich ihr Name oder ein Wert geändert hat. Gleiches kann natürlich in umgekehrter Richtung passieren und wenn in der Tabelle eine Zeile geändert wird, können natürlich gleich mehrere Transitionen betroffen sein.
Mehrfacheinträge werden in der internen Zuordnungstabelle als `Vector`, Einfacheinträge als `Transition` gespeichert. Teile von Methoden, die sich auf Aktionen mit Mehrfacheinträgen beziehen, sind mittels Kommentaren als `multi-xxx` gekennzeichnet, die für Einfacheinträge tragen den Zusatz `single-[Bezeichnung der Funktion]`, beispielsweise `single-add` für das Hinzufügen einer Einzeltransition.

5.2.6.5. SDMV-Mechanismus

Die Interfaces des SDMV-Mechanismus befinden sich im Paket `sdmv` und sind auch im Anhang abgedruckt. Die zugehörigen Implementierungen finden sich alle im Paket `datamanager` (Datenmodell, Datamanager, Adapter) bzw. `action` (alle Akteure wie Syntaxprüfung, Kontextprüfung und Refinement).
Der SDMV-Mechanismus selbst ist in Abschnitt 5.1.3 in aller Ausführlichkeit beschrieben.

5.2.6.6. Sprachunterstützung

Aufgrund der Tatsache, daß die zum Parsen der Textfelder des Automatendokuments verwendete Sprache zu einem späteren Zeitpunkt ausgetauscht werden können soll, wurden sämtliche sprachbezogenen Routinen als reine Interfaces implementiert, welche im Paket `languagesupport` zusammengefaßt sind. Eine konkrete Sprachunterstützung muß diese Interfaces implementieren. Um die Sprachunterstützung zu ändern, reicht es dann, eine einzige Zeile im Konstruktor der Klasse `datamanager.AutomatonDocument` auszutauschen, also beispielsweise statt:

```
language=new Language_ff();
```

einfach

```
language=new Language_xyz();
```

Natürlich muß unter Umständen noch das richtige Paket importiert werden.
Mit wenig mehr Aufwand könnte man auch eine Auswahl der Sprachunterstützung in einen Optionsdialog des Werkzeuges implementieren und so die Sprache zur Laufzeit wechseln.
Das Interface `languagesupport.Language`[3] definiert alle momentan benötigten low-level Methode für die Sprachunterstützung. Das Interface ist so flexibel gestaltet, daß eine möglichst große Zahl von Parsern eingebunden werden kann. Dabei ist es unerheblich, ob es sich um einen echten NLTF-Parser handelt, oder nicht, die Methoden müssen nur richtig implementiert werden. Es werden Parser mit maximal zwei Zerteildurchgängen unterstützt, klassische Parser und inkrementelle Feldparser. FriscoF besitzt nur einen Durchgang und kann nicht inkrementell parsen. Die zu FriscoF gehörende Sprachunterstützung

```
Languagesupport_ff.Language_ff
```

implementiert das Interface also so, daß der erste Zerteilvorgang gänzlich ohne den Parser erledigt wird (es dürfen ja nur syntaktische Prüfungen vorgenommen werden). Der zweite Zerteilvorgang ruft dann den eigentlichen Parser auf (im 2. Durchgang darf auf Kontext geprüft werden). Dies ist notwendig, da FriscoF kein NLTF-Parser ist und die gekapselten Feldkontexte ja erst erstellt werden können, wenn ein Syntaxbaum vorliegt, der aber von FriscoF nicht erstellt werden kann, ohne daß Kontextinformationen vorliegen.
Das Parsen von Feldern mit abgeschlossenem Kontext wird für FriscoF dadurch simuliert, daß bei Übergabe eines jeden Feldes ein geeigneter Kontext erstellt und dann der Parser für dieses eine Feld gestartet wird. Dies ist natürlich sehr ineffizient.
Im Falle eines echten NLTF-Parsers könnte für jedes hinzugefügte Feld ein Eintrag in eine Liste erfolgen. Der Zerteilvorgang müßte erst gestartet werden, wenn die Methode `parsex()` aufgerufen, oder wenn das Ergebnis für ein Feld abgerufen werden würde.

[3]siehe Anhang B.1

Das Interface `languagesupport.LanguageRefinement`[4] definiert eine Schnittstelle für höherwertige Sprachfunktionen, wie sie beispielsweise für den Verfeinerungskalkül benötigt werden. Höherwertige Sprachfunktionen sind praktisch alle Routinen, die als Eingabe Objekte des Interfaces

`languagesupport.AbstractLanguageObject`[5]

akzeptieren und diese weiterverarbeiten. Im Gegensatz zu den niederwertigen Sprachfunktionen, die alle in der Klasse `Language` gesammelt werden sollten, existieren mehrere Klassen für höherwertige Sprachfunktionen, die über den Namen dem Modul, das sie benutzt, zugeordnet werden können sollten, also beispielsweise `LanguageAutomatedReasoning` oder `LanguageCodeGeneration`.
Die Klasse `languagesupport.LanguageError`[6] repräsentiert eine Zuordnungseinheit aus Fehlermeldung und dem Datenobjekt, das den Fehler verursacht hat. Dies ist ein Ersatz für eine Zeilennummer. Über die Methoden des SDMV-Objektes kann dann beispielsweise der Fehlerverursacher farbig hervorgehoben werden.

5.2.6.7. Konfiguration der Benutzerschnittstelle

Während der Implementierung kam es immer wieder zu Schwierigkeiten, weil die Parts der Eingabeeditoren und Hilfswerkzeuge keine Möglichkeit bereitstellten, die von ihnen zur Verfügung gestellten Funktionen teilweise zu deaktivieren.
Bei der Entwicklung des Diagrammeditors wurde zwar eine Konfigurationsmöglichkeit implementiert, jedoch war diese dafür gedacht bei der Entwicklung eines neuen Typs von Diagrammeditor programmtechnisch Funktionen zu deaktivieren. Dieser Mechanismus versagt aber teilweise, wenn diese Konfiguration zur Laufzeit vorgenommen und öfter geändert wird, da keine Zugriffskontrolle implementiert wurde.
Aus diesem Grund wurde für diese Werkzeug ein Mechanismus entwickelt, der auch in anderen Parts zum Einsatz kommen kann. Die zugehörigen Klassen befinden sich im Paket `configrules` und leiten sich ab von:

`Rule`[7]

bzw.

`RuleController`[8]

[4] siehe Anhang B.1
[5] siehe Anhang B.1
[6] siehe Anhang B.1
[7] siehe Anhang B.2
[8] siehe Anhang B.2

Eine Regel korrespondiert mit einer Funktion. Eine Regel kann gesperrt und wieder freigegeben werden. Der `RuleController` überwacht alle Regeln und paßt bei einer Änderung die betroffenen Funktionen und die zugehörige Benutzerschnittstelle an.
Als Beispiel ist die Regel `BooleanRule`[9] implementiert, mit der sich eine Funktion an bzw. abschalten läßt. Es sind aber auch andere Regeln denkbar, mit denen sich Funktionalität feiner konfigurieren läßt.
Eine Besonderheit ist, daß ein Modul, das eine bestimmte Konfiguration in einem anderen Part fordert, über den Sperrmechanismus dafür sorgen kann, daß kein anderer die Konfiguration verändert.

5.2.7. Aktueller Stand der Entwicklung

Der erste Entwicklungsschritt dieser Komponente ist abgeschlossen. In weiteren Schritten werden dann möglicherweise neue Funktionen des OEF implementiert, sobald diese zur Verfügung stehen. Auf der Wunschliste ganz oben steht ein vernünftiger Speichermechanismus. Die momentan verwendete Serialisierung hat den gewaltigen Nachteil, daß zwischen verschiedenen Versionen des Werkzeuges keine Übernahme der gespeicherten Daten möglich ist, sobald sich ein kritischer Bereich ändert.
Dieses Problem kann aber erst vernünftig angegangen werden, wenn für das OEF ein einheitliches Speicherverfahren festgeschrieben wird.
Für einen Datenaustausch über die Grenzen eines Dokumentes hinweg ist es ebenfalls noch zu früh. Auch hier muß abgewartet werden, in welche Richtung sich das Framework entwickelt.
In nächster Zeit wird als weiteres Aktionsmodul ein Quelltextgenerator implementiert werden, der das Automatendokument in eine Programmiersprache übersetzt.

5.3. Die periphere Komponente 'Proof-Manager'

Der Beweismanager ist momentan nur eine Designstudie ohne großartige Funktionalität.

5.3.1. Datenstrukturen - Beschreibung der Klassen

Das Paket `proofmanager` enthält alle Klassen die zum Beweismanager gehören:

- `ProofManager`

  Der Part.

[9] siehe Anhang B.2

- `ProofManagerUI`
  Die Benutzerschnittstelle.

5.3.2. Technische Daten

Archiv:	jar-File
Klasse:	ProofManager
Klassifizierung:	Documentcontroller
Bezeichnung:	Proof Manager
Version:	1.0
Author(en):	Michael Fahrmair

5.3.2.1. Systemvoraussetzungen

Hardware

CPU:	Pentium ab 200MHz, PPC 604 ab 160MHz, UltraSparc
Hauptspeicher:	ab 32M, empfohlen für das ganze OEF: 64M
Graphikkarte:	schnell, mind. 4M
Monitor:	mind. 17 Zoll
Festplattenplatz:	50k, das ganze OEF inkl. Dokumentation: mind. 50M

Betriebssysteme

Windows95/NT:	inkl. Service-Pack 3 (getestet)
Solaris für Sparc:	(getestet)
Linux:	ab Kernel 2.0.30 (nicht getestet)
MacOS:	ab Version 8.0 (nicht getestet)

Java VM und Klassenbibliotheken

JDK-Version:	ab 1.1.5 (bis 1.1.6 getestet)
JFC-Version:	ab 1.0.1 (bis 1.0.2 getestet)
OEF-Version:	ab 1.2.1

5.3.2.2. Java Packages

JDK: `java.awt`
`java.awt.event`
`java.io`
`java.util`

Swing: `com.sun.java.swing`
`com.sun.java.swing.border`
`com.sun.java.swing.event`
`com.sun.java.swing.text`

OEF: `oef.connection`
`oef.parthandler`
`oef.parthandler.interfaces`
`oef.printing`
`oef.scripting`
`oef.session`

5.3.2.3. Latex Packages

fancyvrb

5.3.2.4. Externe Ressourcen

Keine.

5.3.2.5. Unterstützte Interfaces

Aus `oef.parthandler`

Interface	**Anmerkung**
`PartViewer`	Unterstützt, da vom OEF-System gefordert.
`PartEditor`	ja
`PartWithClipBoard`	-
`PartWithConnections`	-
`PartWithHelp`	-
`PartWithMenus`	-
`PartWithPreferences`	-
`PartWithPrinting`	-
`PartWithScaling`	-
`PartWithScriptRecording`	ja
`PartWithScripting`	ja
`PartWithToolBars`	-
`PartWithUndo`	ja
`PartWithSelection`	-

5.3.3. Aktueller Stand der Entwicklung

Der Beweismanager besteht momentan nur aus einer sehr primitiven Datenverwaltung und einer Benutzerschnittstelle.
In einem weiteren Projekt sollte möglichst ein Anschluß an einen automatischen Beweiser implementiert werden.

5.4. Die periphere Komponente ‘FriscoF’

`FriscoF` ist der Sonderfall eines OEF-Parts ohne Benutzerschnittstelle, d.h. es handelt sich lediglich um eine Klassenbibliothek die einen kompletten Parser enthält. Dazu gehört eine Implementierung des in Abschnitt 5.2.6.6 beschriebenen Interfaces, das die Komponente `STDAssistant` mit sprachspezifischen Routinen versorgt.

5.4.1. Architektur

Der Parser besteht aus der Parserumgebung, die alle zum Parsen notwendigen Module sowie den abstrakten Syntaxbaum beherbergt und einer Schnittstelle.
Die Schnittstelle ermöglicht den Zugriff von anderen Komponenten auf die Ergebnisse des Parsvorganges. Dies war nötig, da die ursprüngliche Implementierung des Parsers alle Ergebnisse streng abschirmte.

5.4.2. Kommunikationsfluß

Kommunikation zwischen Parser und Hauptwerkzeug findet ausschließlich über das Sprachunterstützungsinterface (siehe auch Abschnitt 5.2.6.6) statt. Die aus den Textfeldern des Automatendokuments extrahierte Sprachinformation wird für den Parser aufbereitet und an diesen weitergeleitet. Ergebnisse werden in Form von abstrakten Sprachelementen an das Werkzeug zurückgegeben und dort gespeichert. Jede weitere Verarbeitung, beispielsweise die Erzeugung und Ausgabe von Beweisverpflichtungen in der Syntax der verwendeten Sprache, findet dann auf Basis dieser abstrakten Sprachelemente statt.

5.4.3. Datenstrukturen - Beschreibung der Klassen

Das Paket `parsing` enthält die Klasse `FriscoParser`, welche den Zugriff auf den Parser steuert. Die eigentlichen Klassen des Parsers sind teilweise in [15] beschrieben.

5.4.4. Sprachunterstützung

Die Sprachuntertützung für FriscoF kennt folgende abstrakte Sprachelemente:

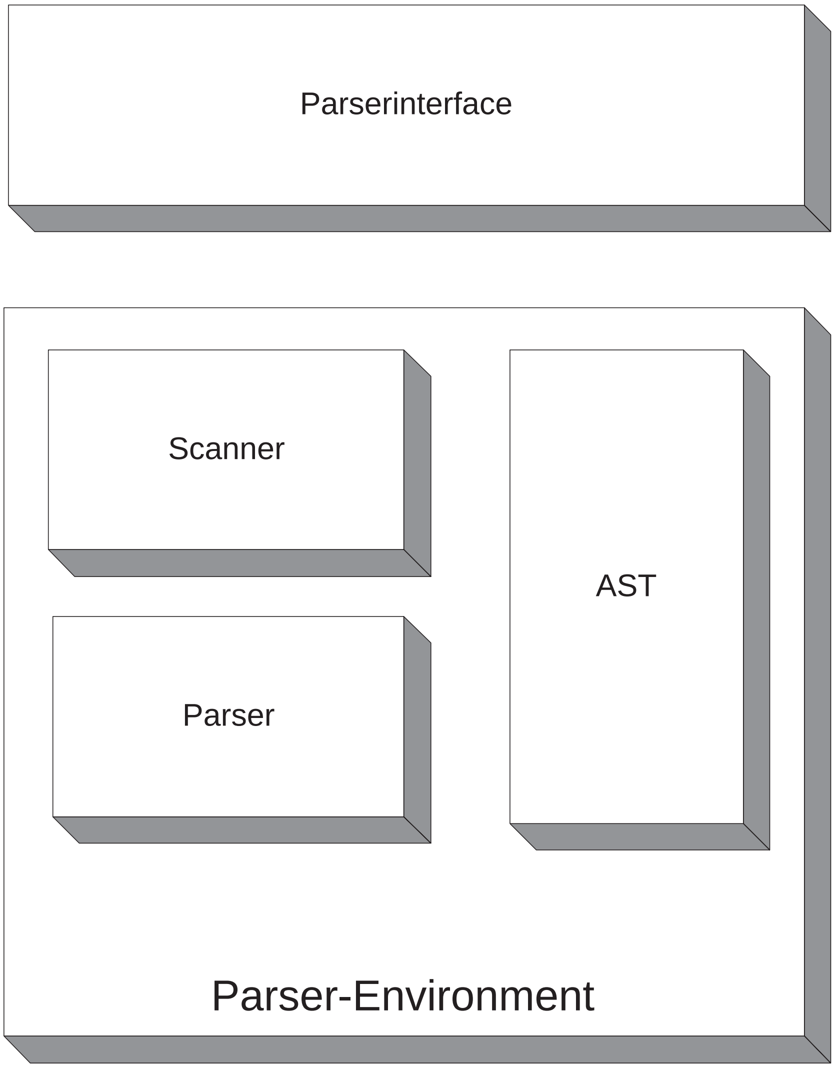


Abbildung 5.13.: Architektur des FriscoF-Parsers

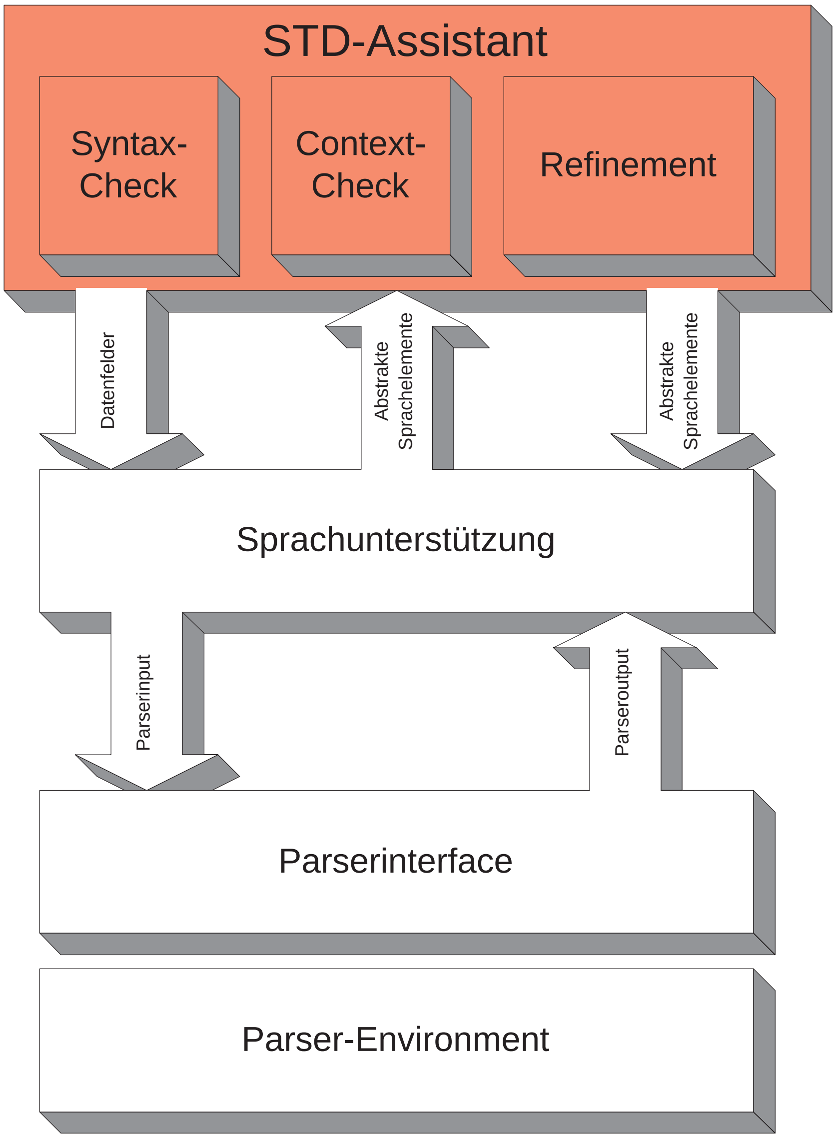


Abbildung 5.14.: Kommunikation STDA - Parser

- Axiom_Abstract

  Speichert ein Axiom, das aus mehreren benannten Bedingungen bestehen kann. Beweisverpflichtungen werden als Axiom gespeichert.

- ClassImportRef_Abstract

  Enthält eine Aufzählung aller importierten Typen für benutzte Objektreferenzen (Eingabe aus dem Abschnitt 'TYPE' der Deklarationen.

- Condition_Abstract

  Speichert ein Prädikat. Dabei kann es sich um ein Zustandsprädikat, eine Vor- oder Nachbedingung aber auch um eine Zeile einer Beweisverpflichtung handeln.

- Declaration_Abstract

  Enthält eine Aufzählung aller deklarierten Attribute (Eingabe aus dem Abschnitt 'ATTRIBUTES' der Deklarationen.

- MethodDeclaration_Abstract

  Enthält eine Liste aller deklarierten Ein- und Ausgabemethoden (Eingabe aus dem Abschnitt 'IN' bzw. 'OUT' der Deklarationen.

- StatePattern_Abstract
- TransitionInputPattern_Abstract
- TransitionOutput_Abstract

Die Sprachuntertützung besteht aus den Klassen `language_ff` und `languageRefinement_ff`, welche die Sprachunterstützungsinterfaces `language` und `languageRefinement` für die Sprache FriscoF implementieren.
In `language_ff` sind Basissprachfunktionen realisiert, die zum Betreiben der Syntax- und Kontextprüfung notwendig sind.
Die Klasse `languageRefinement_ff` enthält die Realisierung erweiterter Sprachfunktionen für die Verfeinerung, wie beispielsweise das Generieren von Beweisverpflichtungen.

5.4.5. Technische Daten

Archiv:	jar-File
Klasse:	-
Klassifizierung:	Library
Bezeichnung:	FriscoF
Version:	1.0
Author(en):	Christian Lesny/Michael Fahrmair

5.4.5.1. Systemvoraussetzungen

Hardware

CPU:	Pentium ab 200MHz, PPC 604 ab 160MHz, UltraSparc
Hauptspeicher:	ab 32M, empfohlen für das ganze OEF: 64M
Graphikkarte:	schnell, mind. 4M
Monitor:	mind. 17 Zoll
Festplattenplatz:	368k, das ganze OEF inkl. Dokumentation: mind. 50M

Betriebssysteme

Windows95/NT:	inkl. Service-Pack 3 (getestet)
Solaris für Sparc:	(getestet)
Linux:	ab Kernel 2.0.30 (nicht getestet)
MacOS:	ab Version 8.0 (nicht getestet)

Java VM und Klassenbibliotheken

JDK-Version:	ab 1.1.5 (bis 1.1.6 getestet)
JFC-Version:	ab 1.0.1 (bis 1.0.2 getestet)
OEF-Version:	ab 1.2.1

5.4.5.2. Java Packages

JDK: `java.io`
`java.lang`
`java.math`
`java.util`

5.4.5.3. Latex Packages

Keine.

5.4.5.4. Externe Ressourcen

Keine.

5.4.5.5. Unterstützte Interfaces

Aus `oef.parthandler`

Interface	**Anmerkung**
`PartViewer`	-

```
PartEditor                 -
PartWithClipBoard          -
PartWithConnections        -
PartWithHelp               -
PartWithMenus              -
PartWithPreferences        -
PartWithPrinting           -
PartWithScaling            -
PartWithScriptRecording    -
PartWithScripting          -
PartWithToolBars           -
PartWithUndo               -
PartWithSelection          -
```

5.4.6. Implementierungsbeschreibung

An der ursprünglichen Implementierung wurde bis auf die öffentliche Freigabe einiger Methoden und Attribute nichts geändert.
Es wurde eine Klasse (`parsing.FriscoParser`) hinzugefügt, welche den Zugriff auf einige Ergebnisse des Parsens ermöglicht.

5.4.7. Stand der Entwicklung

Die rein zu Demonstrationszwecken eingesetzte Sprache FriscoF ist in der momentanen Fassung weniger für den Einsatz in Automatendokumenten geeignet. Der zugehörige Parser wurde außerhalb des OEF-Projektes für den klassischen Einsatz zum Parsen eines zeilenorientierten Quelltextes mit zusammenhängendem Kontext entwickelt.

5.5. Die periphere Komponente ‘StatTable’

Die Komponente `StatTable` entspricht größtenteils der OEF-Standardkomponente `SimpleTableEditor`, welche in [30] ausführlich beschrieben ist.

5.5.1. Stand der Entwicklung

Die Komponente `StatTable` ist eine vorübergehende Modifikation der Komponente `SimpleTableEditor`, in der die Möglichkeit deaktiviert wurde, Spalten zu löschen oder einzufügen. Die Komponente `StatTable` sollte auf keinen Fall weiterentwickelt, sondern sobald wie möglich durch eine offizielle Tabellenkomponente, oder einen Nachfolger von `SimpleTableEditor` mit konfigurierbarer Benutzerschnittstelle ersetzt werden.

5.6. Die periphere Komponente 'SimpleTextEditor'

Die Komponente `SimpleTextEditor` gehört zu den OEF-Standardkomponenten und ist in [30] ausführlich beschrieben.

5.6.1. Stand der Entwicklung

Die Komponente `SimpleTextEditor` wird ohne Änderungen zur übergangsweisen Eingabe der Deklarationsinformationen benutzt.
Sobald Werkzeuge für die anderen Dokumentarten aus [33] bereitstehen und das OEF um einen Mechanismus für dokumentübergreifende `Connections` erweitert wurde, ist der Part 'Declarations' nicht mehr länger Teil des Automatendokumentes. Der zugehörige SDMV-Adapter `DeclData_Access` muß dann komplett neu erstellt werden, so daß ein entsprechendes Connection-Interface verarbeitet werden kann.

6. Schlußbetrachtungen

Dieses Kapitel beschließt die Arbeit mit einer kritischen Bewertung der realen Anwendbarkeit des Werkzeugs sowie der darin implementierten Entwicklungsmethodik und einer kurzen Analyse über die Chancen und Möglichkeiten zur Etablierung der Werkzeugplattform OEF im allgemeinen und des Werkzeuges STDA im speziellen auf dem industriellen, d.h. nichtuniversitären Markt.

6.1. Praktische Bedeutung

Während der Entwicklung der Beispiele für diese Arbeit, die möglichst reale Anwendungen widerspiegeln sollten, hat sich gezeigt, daß obwohl sich die Theorie möglichst nahe an etablierten Verfahren orientiert, die der mathematischen Fundierung zugrundeliegende Philosophie sich auf den ersten Blick stark von derzeit in der praktischen Programmentwicklung verbreiteten Denkprozessen abhebt, die maßgeblich durch die Möglichkeiten und Anforderungen der am weitesten verbreiteten Hochsprachen geprägt sind. Die größte Umstellung scheint die Tatsache zu erfordern, daß sämtlicher Nachrichtenaustausch zwischen Objekten asynchron stattfindet, d.h. Methodenaufrufe keinen Rückgabewert erwarten können. Bei genauerer Betrachtung stellt man jedoch fest, daß auch in moderneren Programmiersprachen asynchrone Konzepte auf dem Vormarsch sind, z.B. die Kommunikationskonzepte für verteilte Komponenten in Java. Einige neue Konzepte des Verfahrens bedürfen aber einer genaueren Betrachtung und sorgfältiger Analyse der Auswirkung auf den praktischen Einsatz.
Das Fehlen synchroner Methodenaufrufe führt zunächst dazu, daß nicht automatisch gesichert ist, daß alle Auswirkungen eines Methodenaufrufes mit dem Beginn der Verarbeitung der nächsten Instruktion bereits erfolgt sind. Ein Methodenaufruf kann z.B. die Daten des aufgerufenen Objektes verändern. Würde nun das aufrufende Objekt sofort nach Absetzen des Methodenaufrufes auf die Daten des aufgerufenen Objektes zugreifen, kann nicht sichergestellt werden, ob die Daten bereits aktualisiert wurden. Die Daten der Objekte müssen also stärker gekapselt werden, d.h. es muß generell verhindert werden, daß ein Objekt auf die Attribute eines anderen direkt zugreift (z.B. über ein Attribut, das als `public` deklariert wurde). Der Umstand, daß nun nicht mehr auf die Attribute anderer Objekte zugegriffen werden kann, da auch keine Methodenaufrufe mit

synchronem Rückgabewert mehr existieren führt dazu, daß auf externe Attribute nur indirekt zugegriffen werden kann, indem zunächst eine Nachricht (Methodenaufruf) an das andere Objekt geschickt wird, mit der Bitte, die Daten eines bestimmten Attributes über einen vorbestimmten Eingabekanal (eigene Methode) zu senden, sobald es bereit dazu ist. Dies führt dazu, daß Objekte einen aktiveren Charakter erhalten, also nur noch auf Ereignisse wie die Änderung eines bestimmten Wertes reagieren. Teilweise kann es sinnvoll sein, Attribute anderer Objekte zu spiegeln, falls ein häufiger Zugriff notwendig ist. Durch den Wegfall synchroner Aufrufe ist es auch nicht mehr einfach möglich, innerhalb eines Objektes häufig benötigte Arbeitsschritte in eigenen Methoden zusammenzufassen. Statt dessen muß die Arbeit u.U. in ein eigenes Objekt ausgelagert werden, oder als Funktion bzw. Operator funktional formuliert werden, d.h. es kann damit nur auf den jeweils spezifizierten Parametern und nicht auf den restlichen Attributen gearbeitet werden.

Einiges Umdenken gegenüber der Entwicklung in klassischen Hochsprachen ist also dennoch gefordert. Es gilt nämlich, die Aufteilung einer Aufgabe so geschickt zu lösen, daß in den einzelnen Klassen möglichst wenig Äquivalenzklassen von Zwischenzustände eingeführt werden müssen, in denen Zeit damit verbracht wird, abhängig von einem momentanen Bearbeitungszustand auf zusätzlich angefragte Daten von anderen Klassen zu warten (d.h. ständig auf Attributwerte anderer Klassen Bezug zu nehmen), da dies die Effizienz und Übersichtlichkeit stark einschränkt.

Diese offensichtlichen Nachteile sind jedoch zur Zeit Gegenstand weiterer Forschungsarbeit und es ist damit zu rechnen, daß die in dieser Arbeit verwendete sehr strikte Methodik in einigen Punkten erweitert wird, um doch synchrone Nachrichten zuzulassen.

Eine weitere Neuerung ist, daß kein Konstruktoraufruf mit parametrisierter Initialisierung existiert. Instanzen werden erzeugt (oder wurden - je nach Modellvorstellung. Praktisch hat es keine Auswirkung, ob man sich vorstellt, daß eine unendliche Anzahl von Instanzen vorerzeugt wurde und dann ein Identifikator aus dieser Menge entfernt wird, wenn auf eine Instanz neu Bezug genommen wird, oder man die Instanz erst erzeugt, wenn ein neuer Identifikator angefordert wird) und mit festgelegten Werten initialisiert. Wünscht man eine Initialisierung mit Parametern, muß man einen Initialisierungszustand einführen, der im Anschluß eine Initialisierungsnachricht mit Parametern akzeptiert und erst dann seine normale Arbeit aufnimmt.

6.2. Etablierungsstrategien für den industriellen Einsatz

Die in Abschnitt 6.1 erwähnte notwendige Umstellung etablierter Denkprozesse und Entwurfsmuster führt dazu, daß das Werkzeug STDA nicht ohne Schwierigkeiten außerhalb eines eng begrenzten universitären Bereiches etabliert werden kann. Auf eine automatische Übernahme in den industriellen Bereich durch abgehende Hochschulabsolventen über eine breitere Etablierung im universitären Bereich zu hoffen, ist die falsche Strategie. Zum einen besitzen beide Bereiche eine von unterschiedlichen Zielsetzungen geprägte Eigendynamik, zum anderen dürfte eine industrielle Etablierung wesentlich leichter zu erreichen und lohnender sein, als eine ausreichend große Verbreitung im Bildungsbereich. Viele Kriterien für einen Einsatz im Bildungsbereich sind für den industriellen Einsatz von untergeordneter Bedeutung, unerheblich oder sogar kontraproduktiv. Es ist also wichtig, von vornherein industrielle Bedürfnisse bei der Weiterentwicklung des Werkzeuges zu berücksichtigen und notfalls vorne anzustellen. Dabei sollten gleichzeitig Maßnahmen zu einer schrittweisen Vergrößerung des Bekanntheitsgrades unternommen werden.
Regionale Messen und Veranstaltungen sind dann ein geeigneter Rahmen für Präsentationen, bei denen nach einem Partner für Feldstudien gesucht werden kann. Damit hätte man den ersten Schritt zur Etablierung getan.
Die Akzeptanz des Werkzeuges im industriellen Einsatz steht und fällt natürlich mit der Einstellung gegenüber der zugehörigen Werkzeugplattform OEF. Über das OEF bietet sich die einmalige Möglichkeit einer schleichenden Etablierung. Wenn es gelingt, das OEF zunächst über den reinen Einsatz als Dokumentationswerkzeug zu positionieren, würde es wesentlich leichter fallen, radikalere und kompliziertere Werkzeuge wie STDA als sogenanntes Zubehör einzuführen.
Dazu müßten die Dokumentationsfähigkeiten des OEF aber noch stark verbessert werden, die Bereiche Repository und Drucklayout sind momentan zwar für den universitären aber nicht für den industriellen Einsatz geeignet.

6.3. Ausblick

Insgesamt zeigt der in dieser Arbeit beschriebene Prototyp, daß ein wesentlicher Schritt in die richtige Richtung gemacht wurde. Es ist aber davon auszugehen, daß vor einer automatisierten Unterstützung eines derartigen Konzepts eine durch Textbücher unterstützte Verbreitung von solchen Verfeinerungsregeln notwendig ist. Einige erste Ansätze hierzu sind bereits zu finden.

Literaturverzeichnis

[1] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings, 1994.

[2] G. Booch, J. Rumbaugh, and I. Jacobson. The Unified Modeling Language for Object-Oriented Development, Version 1.1, 1997.

[3] Wilfried Brauer. *Automatentheorie: eine Einführung in die Technik endlicher Automaten*. Teubner, 1984.

[4] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development - The Fusion Method*. Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1994.

[5] S. Cook and J. Daniels. *Designing Object Systems*. Prentice Hall, 1994.

[6] Michael Fahrmair. Frisco Diagram Editor, Fopra. Master's thesis, Technische Universität München, 1997. www4.informatik.tu-muenchen.de/ syslab/ frisco/ fde/ .

[7] James Gosling and Henry McGilton. *The Java Language Environment – A White Paper, www.javasoft.com/ beans/ white/ langenv/* . Sun Mircosystems, 1996.

[8] R. Grosu, C. Klein, and B. Rumpe. Enhancing the SysLab system model with state. TUM-I 9631, Technische Universität München, 1996.

[9] R. Grosu, C. Klein, B. Rumpe, and M. Broy. State transition diagrams. Technical Report TUM-I9630, Technische Universität München, 1996.

[10] R. Grosu and B. Rumpe. Concurrent timed port automata. Technical Report TUM-I9533, Technische Universität München, 1995.

[11] Graham Hamilton. *JavaBeans, www.javasoft.com/ beans/ docs/ spec.html.* Sun Mircosystems, 1998.

[12] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[13] J. Hopcroft and J. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie.* Addison-Wesley, 1990.

[14] B. Jonsson. *Compositional Verification of Distributed Systems.* PhD thesis, Uppsala University, Uppsala Sweden, 1987.

[15] Christian Lesny. Sprachbeschreibung und Parserimplementierung der funktionalen Sprache „Frisco F“. Master's thesis, Technische Universität München, 1997.

[16] N. Lynch and E. Stark. A Proof of the Kahn Principle for Input/ Output Automata. *Information and Computation*, 82:81–92, 1989.

[17] Sun Mircosystems. *Important Known JDK Bugs, java.sun.com/ products/ jdk/ 1.1/ knownbugs/* , 1997.

[18] Sun Mircosystems. *Object Serialization Specification, java.sun.com/ products/ jdk/ 1.1/ docs/ guide/ serialization/* , 1997.

[19] Sun Mircosystems. *JAR Guide, java.sun.com/ products/ jdk/ 1.1/ docs/ guide/ jar/* , 1998.

[20] Sun Mircosystems. *Java Documentation, www.javasoft.com/ docs*, 1998.

[21] Sun Mircosystems. *Java Early-Access Documentation, java.sun.com/ products/ jdk/ 1.2/ docs/* , 1998.

[22] Sun Mircosystems. *JDK 1.1.6 Documentation Changes, java.sun.com/ products/ jdk/ 1.1/ docchanges.html*, 1998.

[23] Sun Mircosystems. *JDK 1.1.6 Documentation, java.sun.com/ products/ jdk/ 1.1/ docs/* , 1998.

[24] Sun Mircosystems. *PDF and PS Versions of JDK 1.1 Specifications, java.sun.com/ products/ jdk/ 1.1/* , 1998.

[25] B. Paech and B. Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *FME'94, Formal Methods Europe, Symposium '94*, LNCS 873. Springer-Verlag, Berlin, October 1994.

[26] B. Paech and B. Rumpe. Spezifikationsautomaten: Eine Erweiterung der Spezifikationssprache SPECTRUM um eine graphische Notation. In *Formale Grundlagen für den Entwurf von Informationssystemen*, GI-Workshop, Tutzing 24.-26. Mai 1994 (GI FG 2.5.2 EMISA). Institut für Informatik, Universität Hannover, May 1994.

[27] B. Paech and B. Rumpe. State based service description. In J. Derrick, editor, *Formal Methods for Open Object-based Distributed Systems*. Chapman-Hall, 1997.

[28] D. Perrin. Finite Automata. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume Band B, chapter 1, pages 1–58. Elsevier Publisher Amsterdam, 1990.

[29] J. Philipps and B. Rumpe. Refinement of information flow architectures. In M. Hinchey, editor, *ICFEM'97*. IEEE CS Press, 1997.

[30] Olav Rabe. Open Editor Framework, Systementwicklungsprojekt. Master's thesis, Technische Universität München, 1997. www4.informatik.tu-muenchen.de/ syslab/ frisco/ oef/ .

[31] J. Rumbaugh. *Object-Oriented Modelling and Design*. Prentice Hall, 1991.

[32] B. Rumpe. Verwendung endlicher Automaten zur Implementierung des dynamischen Verhaltens von C++ Objekten. In G. Snelting U. Meyer, editor, *Semantikgestützte Analyse, Entwicklung und Generierung von Programmen*, 9402. Justus-Liebig-Universität Giessen, March 1994.

[33] B. Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, 1996. PhD thesis, Technische Universität München.

[34] B. Rumpe and C. Klein. *Automata with output as description of object behavior*, pages 265–286. Kluwer Academic Publishers, Norwell, Massachusetts, 1996.

[35] Sally Shlaer and Stephen J. Mellor. *Object Lifecycles, Modeling the World in States*. Yourdon Press, 1992.

A. Abkürzungen

OEF Open Editor Framework. OEF ist ein Framework zur Erstellung von Werkzeugen.

JFC Java Foundation Classes. JFC ist eine Erweiterung des Java Abtract-Windowing-Toolkits. Es enthält verschiedene Elemente zur Erstellung einer Benutzerschnittstelle, darunter auch komplexe Komponenten, wie einen Dateiauswahldialog.

HTML HyperText Markup Language. Seitenauszeichnungssprache. HTML ist eine Untermenge von SGML für das WWW.

API Application Programming Interface. Schnittstellenbeschreibung einer Bibliothek.

SDMV Synchronized-Data-Model-View. Erweiterter MV-Mechanismus für austauschbare Komponenten.

MV Model-View. Eine Technik, Daten und Darstellung derselben zu trennen, damit für ein Datenmodell verschiedene Anzeigen programmiert werden können.

FDE Frisco Diagram Editor. Generischer Diagrammeditor für OEF. Dieser Parthandler läßt sich mit wenig Aufwand an fast jeden Diagrammtyp anpassen.

HF Human Factor. Menschlicher Faktor. Damit ist der Risikofaktor Mensch gemeint.

STDA State-Transition-Diagram-Assistent. Ein Werkzeug, das bei der Erstellung von Automatendiagrammen assistiert.

NLTF Nichtlineare-(Quell-)Textfelder. Diagrammattributierungen in Textform, wie sie z.B. in Automatendiagrammen vorkommen. Diese Felder lassen sich nur schwer bis überhaupt nicht in eine lineare Abfolge bringen, ganz im Gegensatz zu klassischem Quelltext für den eine Zeilenabfolge eine lineare Ordnung induziert. Für jede Zeile läßt sich angeben, wer ihr Vorgänger oder Nachfolger ist.

JIT Just In Time. Beliebter Modeausdruck. Bedeutet so viel wie ‘sofort im Anschluß’ an ein gewisses Ereignis erhält man das gewünschte Resultat, ohne Wartezeit.

B. Quelltext

B.1. Languagesupport - Interfaces/Classes

```
package languagesupport;

/**
 * A simple interface that all objects that store
 * language-specific data must implement.<p>
 * Support for a given language is done in a package
 * with the following naming scheme:
 *          languagesupport_[name of language]
 * Usually objects of the AST must implement this interface
 *
 * @author  Michael Fahrmair
 * @version 1.0
 * @see     LanguageError
 * @see     LanguageRefinement
 * @see     Language
 */
public interface AbstractLanguageObject {
}

```

```
package languagesupport;

//JDK-packages
import java.util.Vector;

//STDA-packages
import datamanager.*;

/**
 * This interface contains all low-level language-spezific
 * methods. At the moment only those needed by STDAssistant.
 * Other classes that need low-level language-support
 * should expand this interface by adding their own language-
 * methods
 * @author  Michael Fahrmair
 * @version 1.0
 * @see     LanguageError
 * @see     LanguageRefinement
 * @see     AbstractLanguageObject
 */
public interface Language {
  /**
   * Reset Parser
   */
  public void reset();
  /**
   * Set a header which is parsed before each single field,
```

```
 * e.g. to contribute newly defined types or operators that
 * are valid for all fields
 */
public void setPrelude(String data);
/**
 * Define referenced class-types (this is not part of header)
 */
public void addClassImportRef(String data, AbstractSTDDocumentObject source);
/**
 * Define attributes-signatures (this is not part of header)
 */
public void addAttributeDecl(String data, AbstractSTDDocumentObject source);
/**
 * Define signatures of all input-methods (this is not part of header)
 */
public void addInMethodDecl(String data, AbstractSTDDocumentObject source);
/**
 * Define signatures of all output-methods  (this is not part of header)
 */
public void addOutMethodDecl(String data, AbstractSTDDocumentObject source);
/**
 * Add a field containing state-pattern data and supply correct context
 * (referenced objects, attributes but no methods)
 */
public void addStatePattern(String data, AbstractSTDDocumentObject source,
                            AttributeDeclarations ads, ObjectDeclarations ods);
/**
 * Add a field containing state-condition and supply correct context
 */
public void addStateCondition(String data, AbstractSTDDocumentObject source,
                              AttributeDeclarations ads, ObjectDeclarations ods,
                              StatePattern sp);
/**
 * Add a field containing transition-input-pattern and supply correct context
 */
public void addTransitionInputPattern(String data, AbstractSTDDocumentObject source,
                                      AttributeDeclarations ads, InMethodDeclarations imds,
                                      ObjectDeclarations ods);
/**
 * Add a field containing transition-output and supply correct context
 */
public void addTransitionOutput(String data, AbstractSTDDocumentObject source,
                                AttributeDeclarations ads, InMethodDeclarations imds,
                                OutMethodDeclarations omds, ObjectDeclarations ods,
                                StatePattern sp1, StatePattern sp2,
                                TransitionInputPattern ip);
/**
 * Add a field containing transition-precondition and supply correct context
 */
public void addTransitionPrecondition(String data, AbstractSTDDocumentObject source,
                                      AttributeDeclarations ads, InMethodDeclarations imds,
                                      ObjectDeclarations ods, StatePattern sp,
                                      TransitionInputPattern ip);
/**
 * Add a field containing transition-postcondition and supply correct context
 */
public void addTransitionPostcondition(String data, AbstractSTDDocumentObject source,
                                       AttributeDeclarations ads, InMethodDeclarations imds,
                                       OutMethodDeclarations omds, ObjectDeclarations ods,
                                       StatePattern sp1, StatePattern sp2,
                                       TransitionInputPattern ip,TransitionOutput op);

public void addInitialOutput(String data, AbstractSTDDocumentObject source,
                             AttributeDeclarations ads,
                             OutMethodDeclarations omds, ObjectDeclarations ods,
```

```
                              StatePattern sp2);
  public void addInitialPostcondition(String data, AbstractSTDDocumentObject source,
                                     AttributeDeclarations ads,
                                     OutMethodDeclarations omds,
                                     ObjectDeclarations ods, StatePattern sp2,
                                     TransitionOutput op);
  /**
   * Start pass-1 of parsing (syntax-checking)
   */
  public void parse1();
  /**
   * Start pass-2 of parsing (context-checking)
   */
  public void parse2();

  /**
   * get back syntax-errors
   */
  public Vector getSyntaxErrors();
  /**
   * get back context-errors
   */
  public Vector getContextErrors();

  /**
   * get back AST of header
   */
  public AbstractLanguageObject getPrelude(String data);
  /**
   * get back AST of referenced class-types
   */
  public AbstractLanguageObject getClassImportRef(AbstractSTDDocumentObject source);
  /**
   * get back AST of attribute-signatures
   */
  public AbstractLanguageObject getAttributeDecl(AbstractSTDDocumentObject source);
  /**
   * get back AST of input-method-signatures
   */
  public AbstractLanguageObject getInMethodDecl(AbstractSTDDocumentObject source);
  /**
   * get back AST of output-method-signatures
   */
  public AbstractLanguageObject getOutMethodDecl(AbstractSTDDocumentObject source);
  /**
   * get back AST of specified state-pattern
   */
  public AbstractLanguageObject getStatePattern(AbstractSTDDocumentObject source);
  /**
   * get back AST of specified state-condition
   */
  public AbstractLanguageObject getStateCondition(AbstractSTDDocumentObject source);
  /**
   * get back AST of specified transition-input-pattern
   */
  public AbstractLanguageObject getTransitionInputPattern(AbstractSTDDocumentObject source);
  /**
   * get back AST of specified transition-output
   */
  public AbstractLanguageObject getTransitionOutput(AbstractSTDDocumentObject source);
  /**
   * get back AST of specified transition-precondition
   */
  public AbstractLanguageObject getTransitionPrecondition(AbstractSTDDocumentObject source);
  /**
```

```
   * get back AST of specified transition-postcondition
   */
  public AbstractLanguageObject getTransitionPostcondition(AbstractSTDDocumentObject source);
  /**
   * set level of additional context-checks
   */
  public void setContextCheckLevel(int level);
}

```

```
package languagesupport;

//JDK-packages
import java.util.Vector;

//STDA-packages
import datamanager.*;

/**
 * This interface contains all high-level language-spezific
 * methods for refinement.
 *
 * @author  Michael Fahrmair
 * @version 1.0
 * @see     LanguageError
 * @see     AbstractLanguageObject
 * @see     Language
 */
public interface LanguageRefinement {
 /**
  * Get AST of an Axiom that encapsulates all given conditions
  * coming from VB1-rule along with their names with appropriate context
  */
  public AbstractLanguageObject getVB1Conditions(Vector names, Vector conditions);
 /**
  * Get AST of an Axiom that encapsulates all given conditions
  * coming from VB3-rule along with their names with appropriate context
  */
  public AbstractLanguageObject getVB3Conditions(Vector names, Vector conditions,
                                                 AttributeDeclarations ads);
 /**
  * Get AST of an Axiom that encapsulates all given conditions
  * coming from addS-refinement along with their names with appropriate context
  */
  public AbstractLanguageObject getADDSConditions(Vector names, Vector conditions);
 /**
  * Get AST of an Axiom that encapsulates all given conditions
  * coming from refS-refinement along with their names with appropriate context
  */
  public AbstractLanguageObject getREFSConditions(Vector names, Vector conditions);
 /**
  * Get AST of an Axiom that encapsulates all given conditions
  * coming from addT-refinement along with their names with appropriate context
  */
  public AbstractLanguageObject getADDTConditions(Vector names, Vector conditions,
                                                  AttributeDeclarations ads);
 /**
  * Get AST of an Axiom that encapsulates all given conditions
  * coming from refT-refinement along with their names with appropriate context
  */
  public AbstractLanguageObject getREFTConditions(Vector names, Vector conditions,
                                                  AttributeDeclarations ads);
 /**
  * Get AST of an Axiom that encapsulates all given conditions
```

```
  * coming from refI-refinement along with their names with appropriate context
  */
  public AbstractLanguageObject getREFIConditions(Vector names, Vector conditions,
                                                  AttributeDeclarations ads);
 /**
  * Get AST of an Axiom that encapsulates all given conditions
  * coming from remT-refinement along with their names with appropriate context
  */
  public AbstractLanguageObject getREMTConditions(Vector names, Vector conditions,
                                                  AttributeDeclarations ads);
 /**
  * The following methods should be self-explaining
  */
  public AbstractLanguageObject getEnablednesOfStateCond(AttributeDeclarations ads,
                                                         State s);
  public AbstractLanguageObject getStateSpecializationCond(State state, Vector snew,
                                                           AttributeDeclarations ads,
                                                           InMethodDeclarations imds);
  public AbstractLanguageObject getEnablednesOfTransitionCond(Transition t,
                                                              AttributeDeclarations ads,
                                                              InMethodDeclarations imds);
  public AbstractLanguageObject getTransitionsDisjunctCond(Transition t1,Transition t2 ,
                                                           AttributeDeclarations ads,
                                                           InMethodDeclarations imds);
  public AbstractLanguageObject getTransitionOverlappingCond(Transition transition,
                                                             Vector told,
                                                             AttributeDeclarations ads,
                                                             InMethodDeclarations imds);
  public AbstractLanguageObject getTransitionSpecializationCond(Transition transition,
                                                                Vector told,
                                                                AttributeDeclarations ads,
                                                                InMethodDeclarations imds);
  public AbstractLanguageObject getInitialSpecializationCond(Transition transition, Vector told,
                                                             AttributeDeclarations ads,
                                                             InMethodDeclarations imds);
  public AbstractLanguageObject getRefSTransitionCond(Transition t,AttributeDeclarations ads,
                                                      InMethodDeclarations imds);
}

```

```
package languagesupport;

//JDK-packages
import java.io.*;
//STDA-packages
import sdmv.*;
/**
 * This class implements an error-message that is linked
 * to a specific sdmv-object instead of a line number.
 * @author  Michael Fahrmair
 * @version 1.0
 * @see     Language
 * @see     LanguageRefinement
 * @see     AbstractLanguageObject
 */
public class LanguageError implements Serializable{
  public String             text;
  public AbstractSDMVObject ao;

  public LanguageError(String text, AbstractSDMVObject ao) {
    this.text=text;
    this.ao=ao;
  }
}
```

B.2. configrules - Interfaces/Classes

```
package configrules;

/**
 * A simple interface of a configuration rule.
 * Configuration-rules are used for runtime-adaption of
 * userinterface and functionality of oef-parts.<p>
 * Locking of rules must be implemented in extended classes,
 * usually within the methods that change the value of a rule
 *
 * @author  Michael Fahrmair
 * @version 1.0
 * @see     BooleanRule
 * @see     RuleController
 * @see     RuleException
 * @see     RuleIsLockedError
 */
public abstract class Rule {
  protected Object lock;
  protected String name;

  public Rule(String name) {
    this.name=name;
  }

  /**
   * Manually unlock this rule.
   */
  public void unlock(Object requester) {
    if (this.lock==requester) {
      this.lock=null;
    }
  }
  public String getName() {
    return name;
  }

}
```

```
package configrules;

/**
 * This is a simple on/off configuration rule.
 *
 * @author  Michael Fahrmair
 * @version 1.0
 * @see     Rule
 * @see     RuleController
 * @see     RuleException
 * @see     RuleIsLockedError
 */
public class BooleanRule extends Rule{
  protected boolean status=true;

  public BooleanRule(String name) {
    super(name);
  }
  public BooleanRule(String name, boolean status) {
    super(name);
    this.status=status;
  }
  public boolean setStatus(boolean status, Object requester,
```

```
                            boolean lock) throws RuleIsLockedError{
    if ((this.lock==null)||(this.lock==requester)) {
      this.status=status;
      if (lock) this.lock=requester;
      return true;
    } else throw (new RuleIsLockedError());
  }
  public boolean getStatus() {
    return status;
  }
}
```

```
package configrules;

/**
 * A simple interface of a configuration rule.
 * It can be used to target e.g. rule-events
 *
 * @author  Michael Fahrmair
 * @version 1.0
 * @see     BooleanRule
 * @see     Rule
 * @see     RuleException
 * @see     RuleIsLockedError
 */
public interface RuleController {
  public void ruleChanged (String r, boolean status);
}
```

```
package configrules;

/**
 * This is the superclass of all rule-exceptions.
 *
 * @author  Michael Fahrmair
 * @version 1.0
 * @see     BooleanRule
 * @see     RuleController
 * @see     Rule
 * @see     RuleIsLockedError
 */
public class RuleException extends Exception {
}
```

```
package configrules;

/**
 * This error is caused by a request to configure
 * a locked rule
 *
 * @author  Michael Fahrmair
 * @version 1.0
 * @see     BooleanRule
 * @see     RuleController
 * @see     RuleException
 * @see     Rule
 */
public class RuleIsLockedError extends RuleException {
}

```

B.3. sdmv - Interfaces/Classes

```
package sdmv;

//JDK-packages
import java.util.*;
import java.io.*;

/**
 * This is a basic SDMV-Objekt
 *
 * @author  Michael Fahrmair
 * @version 1.0
 * @see     AbstractSDMVEvent
 * @see     AbstractSDMVEventListener
 * @see     SDMVData_Access
 * @see     SDMVData_AccessEvent
 * @see     SDMVData_AccessEventListener
 */

public class AbstractSDMVObject implements Serializable{
  protected Vector  controls;
  public    String  name;
  public    boolean marked=false;

  public boolean equals(AbstractSDMVObject ao) {
    boolean result=true;
    result =result && (name.equals(ao.name));
    return result;
  }
  public AbstractSDMVObject(String name, SDMVData_Access control) {
    controls=new Vector();
    if (control!=null) addControl(control);
    this.name=new String(name);
  }
  public void highlight() {
    Enumeration e = controls.elements();
    SDMVData_Access control;
    while (e.hasMoreElements()) {
      control=(SDMVData_Access)e.nextElement();
      control.highlight(this);
    }
  }
  public void deHighlight() {
    Enumeration e = controls.elements();
    SDMVData_Access control;
    while (e.hasMoreElements()) {
      control=(SDMVData_Access)e.nextElement();
      control.deHighlight(this);
    }
  }
  public void annotate(String t) {
    Enumeration e = controls.elements();
    SDMVData_Access control;
    while (e.hasMoreElements()) {
      control=(SDMVData_Access)e.nextElement();
      control.annotate(this,t);
    }
  }
  public void deAnnotate() {
    Enumeration e = controls.elements();
    SDMVData_Access control;
    while (e.hasMoreElements()) {
      control=(SDMVData_Access)e.nextElement();
```

```
      control.deAnnotate(this);
    }
  }
  public boolean hasNoName() {
    if (name.length()==0) return true;
    else return false;
  }
  public void addControl(SDMVData_Access control) {
    controls.addElement(control);
  }
  public void removeControl(SDMVData_Access control) {
    controls.removeElement(control);
  }
  public void copyControl(AbstractSDMVObject target) {
    Enumeration e = controls.elements();
    SDMVData_Access control;
    while (e.hasMoreElements()) {
      control=(SDMVData_Access)e.nextElement();
      target.addControl(control);
    }
  }
  public void update(AbstractSDMVObject abs) {
    this.name=abs.name;
  }
  public void mark() {
    marked=true;
  }
  public void demark() {
    marked=false;
  }
  public boolean hasValidController() {
    for (int i=0;i<controls.size();i++) {
      SDMVData_Access da = (SDMVData_Access) controls.elementAt(i);
      if (da.isYetValid(this)) return true;
    }
    return false;
  }
  public String getErrorData() {
    return "";
  }
}
```

```
package sdmv;

//JDK-packages
import java.util.*;

/**
 * This is a basic SDMV-Datamodel event
 *
 * @author  Michael Fahrmair
 * @version 1.0
 * @see     AbstractSDMVObject
 * @see     AbstractSDMVEvent
 * @see     AbstractSDMVEventListener
 * @see     SDMVData_Access
 * @see     SDMVData_AccessEventListener
 */
public class AbstractSDMVEvent extends EventObject {
  public static final int NEW_ELEMENT = 0;
  public static final int UPDATE_ELEMENT = 1;
  public static final int UPDATECANCEL_ELEMENT = 2;
  public static final int DEL_ELEMENT = 3;
  public static final int MARK_ELEMENT = 4;
```

```
  public static final int DEMARK_ELEMENT = 5;
  protected int type;

  protected AbstractSDMVObject obj;
  protected Vector objs;

  public AbstractSDMVEvent(Object o,int type,AbstractSDMVObject obj) {
    super(o);
    this.obj=obj;
    this.type=type;
    this.objs=null;
  }
  public AbstractSDMVEvent(Object o,int type,Vector objs) {
    super(o);
    this.obj=null;
    this.type=type;
    this.objs=objs;
  }
  public int getType() {
    return type;
  }
  public AbstractSDMVObject getElement() {
    return obj;
  }
  public Vector getElements() {
    return objs;
  }
}
```

```
package sdmv;

/**
 * This is a basic SDMV-Datamodel event-listener
 *
 * @author  Michael Fahrmair
 * @version 1.0
 * @see     AbstractSDMVObject
 * @see     AbstractSDMVEvent
 * @see     SDMVData_Access
 * @see     SDMVData_AccessEvent
 * @see     SDMVData_AccessEventListener
 */
public interface AbstractSDMVEventListener extends java.util.EventListener {
  public boolean abstractDataChanged(AbstractSDMVEvent e);
}
```

```
package sdmv;

/**
 * This is a basic SDMV-adapter interface
 *
 * @author  Michael Fahrmair
 * @version 1.0
 * @see     AbstractSDMVObject
 * @see     AbstractSDMVEvent
 * @see     AbstractSDMVEventListener
 * @see     SDMVData_AccessEvent
 * @see     SDMVData_AccessEventListener
 */
public interface SDMVData_Access {
  public void    highlight(AbstractSDMVObject key);
  public void    deHighlight(AbstractSDMVObject key);
  public void    annotate(AbstractSDMVObject key,String t);
```

```
  public void    deAnnotate(AbstractSDMVObject key);
  public void    synchronize_Add(AbstractSDMVObject element);
  public void    synchronize_Del(AbstractSDMVObject element);
  public void    synchronize_Update(AbstractSDMVObject element);
  public void    synchronize_Mark(AbstractSDMVObject element);
  public void    synchronize_DeMark(AbstractSDMVObject element);
  public boolean isYetValid(AbstractSDMVObject key);
}
```

```
package sdmv;

//JDK-packages
import draw.*;
import java.util.*;

/**
 * This is a basic SDMV-adapter event
 *
 * @author  Michael Fahrmair
 * @version 1.0
 * @see     AbstractSDMVObject
 * @see     AbstractSDMVEvent
 * @see     AbstractSDMVEventListener
 * @see     SDMVData_Access
 * @see     SDMVData_AccessEventListener
 */
public class SDMVData_AccessEvent extends EventObject {
  public static final int NEW_ELEMENT = 0;
  public static final int UPDATE_ELEMENT = 1;
  public static final int DEL_ELEMENT = 2;
  public static final int MARK_ELEMENT = 3;
  public static final int DEMARK_ELEMENT = 4;

  protected int type;
  protected AbstractSDMVObject abstractObject1,abstractObject2;
  protected Vector abstractObjects;

  public SDMVData_AccessEvent(Object o,int type, AbstractSDMVObject a,AbstractSDMVObject b) {
    super(o);
    this.type=type;
    this.abstractObject1=a;
    this.abstractObject2=b;
    this.abstractObjects=null;
  }
  public SDMVData_AccessEvent(Object o,int type, Vector objs) {
    super(o);
    this.type=type;
    this.abstractObject1=null;
    this.abstractObject2=null;
    this.abstractObjects=objs;
  }
  public int getType() {
    return type;
  }
  public AbstractSDMVObject getElement1() {
    return abstractObject1;
  }
  public AbstractSDMVObject getElement2() {
    return abstractObject2;
  }
  public Vector getElements() {
    return abstractObjects;
  }
}
```

```
package sdmv;

/**
 * This is a basic SDMV-adapter event-listener
 *
 * @author  Michael Fahrmair
 * @version 1.0
 * @see     AbstractSDMVObject
 * @see     AbstractSDMVEvent
 * @see     AbstractSDMVEventListener
 * @see     SDMVData_Access
 * @see     SDMVData_AccessEvent
 */
public interface SDMVData_AccessEventListener extends java.util.EventListener {
  public boolean dataChanged(SDMVData_AccessEvent e);
}
```