Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Die Aufgabe dieser Diplomarbeit besteht in der Entwicklung einer Entwicklungsumgebung für die Unterstützung des Prototyping. Im folgenden wird diese Aufgabe zuerst in den Kontext eingeordnet, anschließend wird die Aufgabenstellung näher beschrieben. Ein Überblick über den Aufbau der Diplomarbeit wird im letzten Abschnitt dieser Einleitung gegeben.

1.1 Kontext

Die Entwicklung komplexer Software-Systeme nach traditionellen Modellen des Software-Lebenszyklusses birgt einige schwerwiegende Nachteile, welche die Qualität der erstellten Software-Systeme negativ beeinflussen können.

Prototyping ist ein methodischer Zugang, der als Möglichkeit für die Verbesserung der Entwicklung von Software-Systemen vorgeschlagen wurde. Dabei ist Prototyping nicht an Sprachen und Werkzeuge gebunden. Es hat sich allerdings gezeigt, daß Prototyping mit den vorhandenen traditionellen Hilfsmitteln nicht erfolgreich durchführbar ist.

PROSET unterstützt als mengentheoretisch orientierte Prototyping-Sprache die schnelle Konstruktion von ausführbaren Prototypen. Zusätzlich sollen alle Aktivitäten beim Prototyping durch geeignete Werkzeuge innerhalb einer offenen und integrierten Entwicklungsumgebung unterstützt werden.

Eine Prototyping-Umgebung, in deren Mittelpunkt PROSET steht, unterstützt die Manipulation verschiedener Arten von Informationen (z.B. Quellcode, Zwischencode), wobei sich attributierte abstrakte Syntaxbäume zur Repräsentation von Programmen bewährt haben. Sie eignen sich als zentrale Informationsstruktur für viele Entwicklungswerkzeuge, wie z.B. syntaxbasierte Editoren, Analysewerkzeuge, Übersetzer oder Interpretierer.

1.2 Aufgabenstellung

Die Aufgabe dieser Diplomarbeit besteht in der Entwicklung einer offenen und integrierten Entwicklungsumgebung zur Unterstützung des Prototyping. Diese Prototyping-Umgebung soll mit Hilfe des Metaprogrammiersystems des Mjølner BETA-Systems in Form einer grammatikbasierten Umgebung für die Sprache PROSET entwickelt werden. Dies umfaßt die Bearbeitung der folgenden Teilaufgaben.

Attributierte abstrakte Syntaxbäume bilden die zentrale Informationsstruktur für die Werkzeuge der Umgebung. Sie bilden ebenfalls die Schnittstelle für zukünftige Werkzeuge. Eine formale Definition ist daher wünschenswert.

Im Rahmen dieser Arbeit werden die grundlegenden Werkzeuge zur Bearbeitung von PROSET-Programmen entwickelt. Dies sind ein syntaxbasierten Editor, ein Parser, ein Unparser und ein Werkzeug zur Analyse der statischen Semantik. Weitere Werkzeuge zur Analyse von Programmen sind ebenfalls wünschenswert.

Die Komponenten der Umgebung müssen untereinander in mehreren Dimensionen integriert sein: Die Datenintegration erfordert die einheitliche, persistente Verwaltung der Informationsstrukturen. Im Hinblick auf die Präsentationsintegration ist eine einheitliche Benutzungsschnittstelle erforderlich. Die Kontrollintegration dient der Kooperation der Werkzeuge, wobei eine Realisierung auf Basis von ToolTalk untersucht wird.

1.3 Überblick

In Kapitel 2 werden die für das Verständnis der Arbeit relevanten Grundlagen erläutert. Neben einem Überblick über Prototyping und Entwicklungsumgebungen, werden wichtige Eigenschaften der Sprache PROSET erläutert. Zusätzlich werden grundlegende Betrachtungen zur Entwicklung grammatikbasierter Umgebungen durchgeführt und das BETA-Metaprogrammiersystem vorgestellt.

Die folgenden drei Kapitel orientieren sich an den klassischen Phasen des Software-Lebenszyklusses Anforderungsanalyse, Entwurf und Realisierung.

Kapitel 3 befaßt sich mit der Diskussion der Anforderungen an eine Prototyping-Umgebung. Dies geschieht nicht in Form einer formalen Definition, sondern es werden informell sinnvolle und wünschenswerte Eigenschaften zusammengestellt. Nach einer Problembeschreibung werden dazu die verschiedenen Integrationsaspekte und Anforderungen an die Werkzeuge der Umgebung betrachtet.

Das nachfolgende Kapitel 4 beschreibt den Entwurf einer Prototyping-Umgebung für die Sprache PROSET. Dieses Kapitel besteht, neben einem kurzen Überblick in Form eines Grobentwurfs, aus der Beschreibung der einzelnen Komponenten der Umgebung.

Kapitel 5 beschreibt einige wichtige Aspekte der Realisierung der Prototyping-Umgebung mit dem BETA-Metaprogrammiersystem. Die Realisierung umfaßt dabei die Kodierung und Tests. Dabei werden auch Probleme bei der Realisierung angesprochen.

Kapitel 6 enthält eine Zusammenfassung der Ergebnisse der Diplomarbeit und weist auf mögliche Erweiterungen der entwickelten Prototyping-Umgebung hin.

Den Abschluß bilden Installationshinweise für die Prototyping-Umgebung, sowie die PROSET-spezifische Bedienung der Umgebung in Form eines Anhangs.

Konventionen

In diesem Dokument werden englische Begriffe *kursiv* gesetzt. Programmtext der Sprache PROSET wird durch Verwendung einer anderen `Schriftart` kenntlich gemacht. Programmtext der Sprache BETA wird ebenso durch Verwendung *kursiver* Schrift kenntlich gemacht.

2 Grundlagen

In diesem Kapitel werden die zum Verständnis der Arbeit relevanten Grundlagen erläutert. Zunächst wird das Prototyping als methodischer Zugang bei der Software-Entwicklung diskutiert. Die systematische Konstruktion komplexer und qualitativ hochwertiger Software-Systeme erfordert eine geeignete Entwicklungsumgebung. In Abschnitt 2.2 werden verschiedene Ansätze für derartige Umgebungen unterschieden. Insbesondere wird der für diese Arbeit wichtige Begriff einer Prototyping-Umgebung festgelegt. Im Anschluß werden Eigenschaften der Prototyping-Sprache PROSET beschrieben, soweit sie Einfluß auf die Erstellung einer Entwicklungsumgebung haben. Betrachtungen zu Techniken und Werkzeugen zur Erstellung grammatikbasierter Entwicklungsumgebungen, insbesondere die Eigenschaften des BETA-Metaprogrammiersystems, das als Basissystem der Entwicklung der Prototyping-Umgebung zugrunde liegt, bilden den Abschluß des Kapitels.

2.1 Prototyping

Die Entwicklung von Software-Systemen wurde im Laufe der Zeit zu einer Aufgabe mit ständig wachsender Komplexität, die ohne einen organisatorischen und planerischen Rahmen nicht mehr beherrschbar ist. Im Bereich der Informatik hat sich das Forschungsgebiet Software-Technologie entwickelt, das auf die Einführung ingenieurmäßiger Vorgehensweisen in der Software-Entwicklung abzielt. Dort werden Methoden für die systematische Konstruktion komplexer und qualitativ hochwertiger Software-Systeme erarbeitet.

Um den komplexen Prozeß der Software-Entwicklung zu strukturieren und geeignete Vorgehensweisen und Methoden entwickeln zu können, sind schon frühzeitig Modelle für den Software-Lebenszyklus vorgeschlagen worden. Das Wasserfallmodell [GJM91] gestattet die Planung und Steuerung eines Projektes gemäß Standards und Techniken des Ingenieurbereichs. Allerdings birgt dieses Modell einige Nachteile, die wegen ihrer Tragweite nicht unerwähnt bleiben sollen:

- Die Bereiche Spezifikation und Implementation sind vollständig voneinander getrennt, obwohl es in der Praxis viele Abhängigkeiten und gegenseitige Einflüsse gibt.
- Die Interaktion mit dem Benutzer ist beschränkt.
- Eine formale, nicht ausführbare Spezifikation ist für Nichtinformatiker unverständlich.
- Ablauffähiger Code ist erst in einem sehr späten Stadium des Entwicklungsprozesses verfügbar.

Neben dem in streng sequentiellen Phasen ablaufenden Wasserfallmodell sind auch flexiblere Modelle entwickelt worden, beispielsweise das von Boehm in [Boeh86] vorgeschlagene Spiralmodell. Diese moderneren Modelle brechen die strenge Abfolge der Phasen auf, indem inkrementelles Vorgehen und der kontrollierte Wechsel in frühere Phasen vorgesehen wird.

Mangelhafter Dialog zwischen den Entwicklern von Software-Systemen und den Benutzern führt in der Praxis häufig dazu, daß die erstellten Systeme nicht die Forderungen der Benutzer erfüllen.

Prototyping versucht die genannten Nachteile abzuschwächen. Prototyping ist ein methodisches Vorgehen, das als Ergänzung zu den traditionellen Vorgehensweisen bei der Software-Entwicklung verstanden werden kann. **Prototyping** ist nach Floyd [Floy84] eine wohldefinierte Phase im Software-Entwicklungsprozeß, in der ein ablauffähiges Modell (ein **Prototyp**) entwickelt wird, das die entscheidenden Eigenschaften des endgültigen Modells besitzt. Es wird dazu herangezogen, um die Eigenschaften zu überprüfen und das Vorgehen bei der weiteren Entwicklung zu bestimmen. Im folgenden werden die Zugänge zu Prototyping näher beschrieben.

2.1.1 Zugänge zu Prototyping

Das Hauptanliegen des Prototyping, positiven Einfluß auf den Software-Entwicklungsprozeß zu nehmen, kann auf durchaus unterschiedliche Arten erreicht werden. Im folgenden werden drei Aspekte des Prototyping näher betrachtet:

- Die Ziele des Prototyping
- Horizontales und vertikales Prototyping
- Der Prototyp im Software-Entwicklungsprozeß

Detaillierte Diskussionen findet der interessierte Leser in [BKKZ92] und [KLSZ92].

2.1.1.1 Ziele des Prototyping

Neben dem Einsatz in der Analyse des Anwendungsbereiches kann Prototyping auch bei Entwurf und Realisierung von Software-Systemen eingesetzt werden. Im folgenden werden die Ziele genauer betrachtet, die durch den Einsatz von Prototyping erreicht werden sollen.

Exploratives Prototyping dient der Klärung der Problemstellung. Hier werden auf Basis von frühen Ideen Anforderungen erarbeitet. Für die Entwickler besteht während der Prototyping-Phase die Möglichkeit den Anwendungsbereich näher kennenzulernen und idealerweise eine Kommunikationsbasis zwischen Entwicklern und Benutzern zu schaffen.

Experimentelles Prototyping zielt auf die technische Realisierung ab. Durch Experimentieren werden die Anforderungen der Benutzer weiter herausgearbeitet. Des weiteren wird auf der Seite der Entwickler das Augenmerk auf die technische Umsetzung gelegt.

Evolutionäres Prototyping wird als Entwicklungsprozeß verstanden, der die Anwendungsentwicklung fortlaufend begleitet. Die Anwendung soll so an die sich schnell ändernden organisatorischen Anforderungen und weiteren Randbedingungen angepaßt werden. Die Entwickler müssen eine sehr enge Zusammenarbeit mit den Benutzern unterhalten, um die Anwendung fortlaufend zu verbessern.

2.1.1.2 Horizontales und vertikales Prototyping

In der Praxis werden häufig zwei unterschiedliche Vorgehensweisen eingesetzt, die mit der Entwicklung eines Software-Systems in Form von Schichten verbunden sind.

Horizontales Prototyping beschreibt die Realisierung bestimmter Schichten eines Anwendungssystems in Form eines Prototyps. Weit verbreitet ist beispielsweise das Oberflächenprototyping zur Gestaltung der Mensch-Maschine-Schnittstelle. Allerdings können auch andere Schichten der Anwendung durch horizontales Prototyping realisiert werden.

Vertikales Prototyping beschreibt dagegen die Realisierung eines Teils der Anwendung über alle Schichten hinweg als Prototyp.

2.1.1.3 Der Prototyp im Software-Entwicklungsprozeß

Die Konstruktion von Prototypen kann Einfluß auf unterschiedliche Aktivitäten des Software-Entwicklungsprozesses nehmen. So können Einflüsse bereits bei der Projektakquisition, in der Analysephase oder auch bei Entwurf und Realisierung auftreten. Die verschiedenen Einflüsse des Prototypen können als Kriterium zur Klassifikation verschiedener Arten von Prototypen dienen:

- Ein **Demonstrationsprototyp** dient der Akquisition von Projekten. Mit seiner Hilfe kann die prinzipielle Machbarkeit eines Projektes untermauert werden, oder es kann frühzeitig gezeigt werden, daß die Handhabung den Vorstellungen der Benutzer entspricht. Ein Demonstrationsprototyp ist so weit von einem fertigen Anwendungssystem entfernt, daß der Benutzer zwar die Prinzipien des späteren Produktes erkennen kann, ihm aber gleichzeitig die Beschränkungen des Prototypen selbst erläutert werden können. So werden keine falschen Erwartungen bei den Benutzern erzeugt.
- Der **Prototyp im engeren Sinne** wird während der Analysephase erstellt. Es können in der Regel nur einzelne Aspekte modelliert werden, wie Teile der Benutzeroberfläche oder der Funktionalität. Dies hilft bei der Klärung von Problemen und der Kommunikation mit den Benutzern.
- Der Prototyp als **Pilotsystem** wird durch Erweiterung zu einem fertigen Anwendungssystem ausgebaut. Die strikte Trennung zwischen dem Prototypen und dem Anwendungssystem wird hierbei aufgehoben. Statt dessen wird der Prototyp sukzessive um zusätzliche Teile ergänzt und verbessert.
- Ein **Laborprototyp** dient der Klärung technischer Fragen der Entwickler in Bezug auf das Anwendungssystem. Hierbei werden Fragestellungen bezüglich der Funktionalität oder der Architektur untersucht. Die Benutzer sind an der Bewertung von Laborprototypen nicht beteiligt.

Es hat sich gezeigt, daß für eine erfolgreiche Durchführung des Prototyping geeignete Sprachen und Werkzeuge, sowie neue Organisationsformen entwickelt werden müssen [KLSZ92]. Im folgenden wird der Einfluß der Werkzeuge auf die Software-Entwicklung verdeutlicht.

2.2 Entwicklungsumgebungen

Um den Entwickler von Software innerhalb der einzelnen Phasen des Entwicklungsprozesses über den vollständigen Lebenszyklus von Software zu unterstützen, sind Methoden entwickelt worden, die teilweise auch in Form von Software-Werkzeugen verfügbar sind. Verbreitete Werkzeuge für die Implementierungsphase des Software-Entwicklungsprozesses sind beispielsweise Editoren, Übersetzer für Maschinen- und Hochsprachen oder auch Werkzeuge zur Fehlersuche. Weniger verbreitet sind bisher Werkzeuge für die Analyse- oder die Entwurfsphase. Wünschenswert ist die Entwicklung von Software-Werkzeugen für alle Phasen des Entwicklungsprozesses, die gemeinsam als Entwicklungsumgebung die Entwicklung von Software unterstützen.

In den folgenden Abschnitten werden verschiedene Ansätze von Entwicklungsumgebungen diskutiert. Zunächst werden Software-Entwicklungsumgebung und Programmierumgebung unterschieden. Im Hinblick auf die zu entwickelnde Prototyping-Umgebung werden anschließend spezifische Aspekte für eine Umgebung zur Unterstützung des Prototyping aufgezeigt.

2.2.1 Software-Entwicklungsumgebung

Das Ergebnis des Software-Entwicklungsprozesses besteht nicht nur aus einem ablauffähigen Software-System, sondern auch aus anderen Produkten, die in den verschiedenen Phasen der Entwicklung erstellt werden. Dies umfaßt zum Beispiel das Pflichtenheft, den Architekturentwurf, die Zeit- und anderen Projektpläne oder die Dokumentation für Entwickler und Benutzer. Ebenso wie das ablauffähige System liegen alle weiteren Produkte des Entwicklungsprozesses in Form von Dokumenten vor.

Ein Software-System zur Manipulation dieser verschiedenen zu einem Software-System gehörenden Dokumente bezeichnet man als **Software-Entwicklungsumgebung** (SEU). Hauptziel von SEUs ist es, die Unterstützung der Entwickler bei der Erstellung von Software-Systemen zu verbessern. Durch eine Entlastung von nicht-kreativer Arbeit, sowie durch die Unterstützung bei fehlerträchtiger Arbeit soll eine Produktivitätssteigerung erreicht werden. Ebenso wird auch eine Verbesserung der Qualität der erstellten Software-Produkte angestrebt. Eine Software-Entwicklungsumgebung deckt die Aktivitäten im Zusammenhang mit der Entwicklung von Software in allen Phasen des Lebenszyklusses ab, also neben Entwurfsaktivitäten, Programmerstellung und Test, auch Konfigurations- und Projektmanagement.

Zur Erreichung dieses Zieles sind in der Vergangenheit verschiedene Ansätze gewählt worden. Diese Ansätze reichen von der Zusammenfassung konventioneller Werkzeuge zu einem Werkzeugkasten (Software-Entwicklungswerkbank) über CASE-Systeme und KI-Ansätze (Programmier-Assistenten) bis zur Software-Fabrik, einer generischen Umgebung, die für den aktuellen Kontext konfigurierbar ist [Nagl93].

Seit Beginn der achtziger Jahre haben sich folgende Richtungen bei der Verbesserung von Entwicklungsumgebungen herausgebildet: Neben der Vergrößerung der Ausdrucksstärke und Mächtigkeit von Programmiersprachen wurde verstärkt an der Verbesserung der Zusammenarbeit der am Software-Entwicklungsprozeß beteiligten Werkzeuge durch Integration und inkrementelle Verarbeitung gearbeitet.

2.2.1.1 Integration

Als erster Ansatz wurden einzelne Werkzeuge zu Werkzeugansammlungen zusammengefaßt. Später wurde die weiterhin bestehende Isolierung der Werkzeuge aufgebrochen, sie wurden untereinander **integriert**, so daß verschiedene Werkzeuge kooperativ genutzt werden konnten. Gleichzeitig wurde auch die Mächtigkeit der beteiligten Werkzeuge wie Editoren, Übersetzer, Werkzeuge zur Fehlersuche oder Konfigurations-Management vergrößert. Der für Entwicklungsumgebungen zentrale Begriff der Integration wird genauer in Abschnitt 3.2 betrachtet.

2.2.1.2 Inkrementelle Verarbeitung

Die mit dem Software-Entwicklungsprozeß verbundenen Dokumente haben üblicherweise eine innere Struktur. Sie bestehen aus Teildokumenten, die selbst wiederum verfeinert werden können. Wenn die Struktur eines Dokumentes formal beschrieben werden kann, das Dokument also eine formale Syntax besitzt (z.B. als Beschreibung in der erweiterten Backus-Naur-Form EBNF), dann bezeichnet man die syntaktischen Einheiten der Sprache als **Inkremente**. Eine Verarbeitung von Dokumenten, die sich an der formalen Syntax orientiert, bezeichnet man dann als inkrement-orientiert oder inkrementell.

Als Beispiel sei die Struktur von Benutzerhandbüchern und Programmdokumenten aufgeführt. Benutzerhandbücher setzen sich beispielsweise aus Kapiteln zusammen, diese Kapitel bestehen aus einzelnen Abschnitten. Jeder Abschnitt besteht aus einer Überschrift und einer Folge von Absätzen. Dabei sind Kapitel, Abschnitt, Überschrift und Absatz mögliche Inkremente zur formalen Beschreibung der Struktur von Benutzerhandbüchern. Dagegen bestehen Programmdokumente aus anderen Arten von Inkrementen. Programme sind z.B. zusammengesetzt aus einzelnen Modulen, die unter anderem aus Prozeduren bestehen. Prozeduren selbst bestehen aus Anweisungen und möglicherweise weiteren (lokalen) Prozeduren.

Zwischen Inkrementen können dabei vielfältige Querbezüge existieren. So sind nicht nur Bezüge zwischen Inkrementen desselben Dokumentes denkbar, sondern auch zwischen Inkrementen verschiedener Dokumente, wie z.B. zwischen Entwurfs- und Implementationsdokumenten.

Die Integration der Werkzeuge fördert die Möglichkeit von der traditionellen Arbeitsweise hin zu einer inkrementellen Arbeitsweise überzugehen. Die traditionelle Arbeitsweise ist dabei charakterisiert durch die sequentielle Verarbeitung ganzer Dokumente und der zyklischen Benutzung getrennter Werkzeuge, wie es beispielsweise in dem Zyklus Editieren-Übersetzen-Ausführen der Fall ist. Bei der inkrementellen Verarbeitung werden dagegen die Werkzeuge zeitlich verzahnt ausgeführt, wobei jeweils kleine Änderungen von allen beteiligten Werkzeugen verarbeitet werden. Auf diese Art erreicht man das Aufbrechen der Arbeitszyklen und gewährleistet die Proportionalität zwischen dem Umfang der Änderung und dem Umfang der Neuberechnungen. Somit können kürzere Antwortzeiten erreicht werden, als es mit anderen Ansätzen zur Verkürzung der Verarbeitungszyklen, wie beispielsweise getrennter Übersetzung oder inkrementellem Binden möglich ist.

Inkrementelle Werkzeuge können wesentliche Nachteile der traditionellen Arbeitsweise überwinden helfen: lange Wartezeiten während des Laufs einzelner Werkzeuge entfallen, komplexe Kommandosprachen können durch am Kontext orientierte Kommandos ersetzt werden, das Auffinden von Fehlerquellen in inneren Schritten der Verarbeitung entfällt, die Rückmeldung von Fehlern erfolgt unmittelbar.

2.2.2 Programmierumgebung

Im Gegensatz zu einer Software-Entwicklungsumgebung unterstützt eine **Programmierumgebung** lediglich die Aktivitäten in der Implementierungsphase. Es wird also vorwiegend die Erstellung von Programmdokumenten unterstützt. Eine Programmierumgebung enthält dazu folgende Werkzeuge [ES89]:

Editor:
Der Editor dient der Erstellung und Modifikation der Programme. Die Bandbreite reicht hier von einfachen Texteditoren über Editoren, die ebenfalls textbasiert sind, aber die Entwicklungssprache unterstützen (z.B. durch die textuelle Einfügung von Programmkonstrukten und die Unterstützung bei der Einrückung), bis hin zu syntaxbasierten Editoren, die statt textueller Erstellung eine vollständig programmiersprachenspezifische Unterstützung auf der Basis von Inkrementen bieten.

Übersetzer oder Interpretierer:
Übersetzer und Interpretierer geben dem Entwickler die Möglichkeit der Überprüfung und der Ausführung des erstellten Programms.

Werkzeug zur Fehlersuche:
Die Bereitstellung eines Werkzeuges zur Fehlersuche (*debugger*) ist nicht zwingender Bestandteil einer Programmierumgebung, jedoch kann durch dessen Einsatz die Fehlersuche erheblich vereinfacht werden.

Testwerkzeug:
Auch das Testwerkzeug ist eine optionale Komponente einer Programmierumgebung. Es dient der Unterstützung bei der Validierung von Programmen, durch die Erzeugung von Testdaten, die Durchführung des Tests und die Auswertung der Testergebnisse.

Je nach Architektur der Umgebung können bestimmte Aufgaben unterschiedlichen Werkzeugen übertragen werden. Beispielsweise kann die Sicherstellung der syntaktischen Korrektheit eines Programms entweder durch den Übersetzer/Interpretierer oder bereits durch den Editor erfolgen (entsprechendes Wissen über die Sprache vorausgesetzt). Ebenso kann zur Prüfung der Syntax eines Programms auch ein dediziertes Werkzeug eingesetzt werden.

Die Aufgabe der Entwicklung einer SEU, die den gesamten Entwicklungsablauf unterstützt, ist recht umfangreich. Als ersten Schritt auf dem Weg zu einer SEU bietet es sich daher an, Werkzeugunterstützung für die Implementierungsphase des Software-Entwicklungsprozesses in Form einer Programmierumgebung zu schaffen.

2.2.3 Prototyping-Umgebung

Eine **Prototyping-Umgebung** (PU) ist eine Entwicklungsumgebung, die eine Erstellung von Prototypen im Rahmen der gewählten Zugänge unterstützt. Dabei sollte Wahlfreiheit bei der Auswahl eines Zugangs bestehen. Die PU darf die Entwickler nicht von Anfang an in eine bestimmte Richtung lenken, indem sie bestimmte Zugänge verwehrt. Die Auswahl muß durch die Entwickler fallweise entschieden werden können.

Eine Prototyping-Umgebung sollte dabei die folgenden Kriterien erfüllen: Eine PU muß auf einer möglichst mächtigen Prototyping-Sprache aufbauen, damit z.B. evolutionäres und auch vertikales Prototyping unterstützt werden können. Gefordert ist die schnelle Konstruktion von Prototypen und deren einfache Modifikation durch die Bereitstellung mächtiger Transformationen. Die schnelle Konstruktion wird insbesondere durch die Verarbeitung unvollständiger Spezifikationen unterstützt.

Im Rahmen einer PU muß die Werkzeugunterstützung zumindest in einem ähnlichen Umfang gewährleistet sein, wie dies auch für Programmierumgebungen der Fall ist. Prototyping- und Programmierumgebungen unterstützen jeweils eine Phase des Entwicklungsprozesses. So sind ein Editor und ein Werkzeug zur Programmausführung (Übersetzer oder Interpretierer) unabdingbar für die Erstellung ausführbarer Modelle. Aber auch Unterstützung bei der Evaluation der Prototypen sollte in Form von Werkzeugen gegeben sein.

Der Umfang einer Prototyping-Umgebung kann über den Umfang einfacher Programmierumgebungen weit hinausgehen, wenn für alle Zugänge methodische Unterstützung bereitgestellt wird. So ist über die oben beschriebenen Werkzeuge hinaus die zusätzliche Unterstützung durch weitere Werkzeuge denkbar, wie Übersetzer-Generatoren (z.B. für Kommandosprachen und deren Verarbeitung) oder auf bestimmte Zugänge des Prototyping zugeschnittene Werkzeuge, z.B. Werkzeuge zur Beschreibung von Benutzeroberflächen oder Generatoren für Datenbankzugriffe für die Unterstützung des horizontalen Prototypings.

2.2.4 Architekturarten von Entwicklungsumgebungen

In diesem Abschnitt werden gebräuchliche Architekturarten von Software-Entwicklungsumgebungen anhand des Überblicks von Knudsen et al. [KSM94] skizziert. Zu Beginn werden traditionelle Architekturen vorgestellt (separate Werkzeuge und Pipeline-Architektur), es schließen sich modernere Ansätze an (residente Architektur, Bus- und Client/Server-Architektur und grammatikbasierte Architektur). Besonderes Augenmerk wird hier auf die Unterstützung der Integration durch die Architektur gelegt.

Die Werkzeuge einer Entwicklungsumgebung repräsentieren und manipulieren unterschiedliche Arten von Informationen, wie Programmtexte, Handbücher oder Laufzeitstrukturen. Zusätzlich gibt es eine Reihe von verschiedenen Informationsformen, wie die Position der Schreibmarke eines Editors, die Symboltabelle eines Übersetzers oder die Unterbrechungspunkte eines Werkzeugs zur Fehlersuche, die im Rahmen spezifischer Werkzeuge eine Rolle spielen. Im Hinblick auf Integration ist die gemeinsame Haltung gleicher spezifischer Informationen wünschenswert. Die eigenständige Repräsentation gleicher Informationen birgt die Gefahr von Inkonsistenzen durch unterschiedliche Interpretation seitens verschiedener Werkzeuge.

Aufgrund der Eignung der grammatikbasierten Architektur für die Verarbeitung von Programmen wird in dieser Arbeit die Entwicklung einer Prototyping-Umgebung mit grammatikbasierter Architektur vorgenommen.

Abbildung 1 stellt die verschiedenen im folgenden diskutierten Architekturarten schematisch gegenüber. Dort bezeichnen ovale Symbole die Funktionalität der Werkzeuge, kleine Rechtecke bezeichnen die Repräsentation. Rechtecke mit abgerundeten Ecken dienen der Darstellung von Adreßräumen. Pfeile zwischen den Werkzeugen stellen Informationsfluß dar. Die Kopplung von Werkzeugfunktionalität und -repräsentation wird durch Verbindungslinien dargestellt, deren unterschiedliche Stärke die Enge der Kopplung bezeichnet.

2.2.4.1 Separate Werkzeuge

Das traditionelle Vorgehen bei der Entwicklung von Software ist die Verwendung von separaten Werkzeugen. In Abbildung 1 a) werden diese separaten Werkzeuge dargestellt. Jedes Werkzeug unterhält eine eigene Repräsentation. Die Werkzeuge sind isoliert, es existiert keine Möglichkeit direkte Unterstützung bei dem Austausch von Informationen bereitzustellen. Jedes Werkzeug muß Konvertierungsaufwand betreiben, um Informationen anderer Werkzeuge zu erhalten. Dies scheitert oft an nicht oder nur mangelhaft offengelegten Strukturen der Repräsentationen.

Beispiel für die Benutzung separater Werkzeuge ist die Programmentwicklung mittels Text- oder Programmeditoren, die Verwendung kommandozeilenbasierter Übersetzer und die Benutzung eines

Textverarbeitungssystems zur Erstellung der Dokumentation, wie es z.B. bei Macintosh Rechnern mit Textverarbeitungsprogrammen wie *Word* oder Übersetzern wie *Think C* üblich ist.

2.2.4.2 Pipeline-Architektur

In einer Pipeline-Architektur werden Werkzeuge nacheinander ausgeführt, wobei Informationen an das nachfolgende Werkzeug weitergegeben werden. Dazu müssen zwei aufeinanderfolgende Werkzeuge ein gemeinsames Format zur Kommunikation definieren. Abbildung 1 b) zeigt die schematische Darstellung einer Pipeline-Architektur. Auch hier unterhält jedes Werkzeug eine eigene Repräsentation.

Ein Nachteil dieser Architektur zeigt sich in der Praxis oftmals bei der Abstimmung des Austauschformates: Geringfügige Unterschiede des bereitgestellten und des erwarteten Formates können dazu führen, daß keine Kommunikation zwischen zwei Werkzeugen möglich ist, sie können nur noch isoliert benutzt werden. Die durch Pipeline-Architekturen erzeugten Verarbeitungszyklen können durch entstehende lange Durchlaufzeiten negativen Einfluß auf die Produktivität haben. Die Lokalisierung von Fehlern in inneren Schritten der Pipeline wird erschwert.

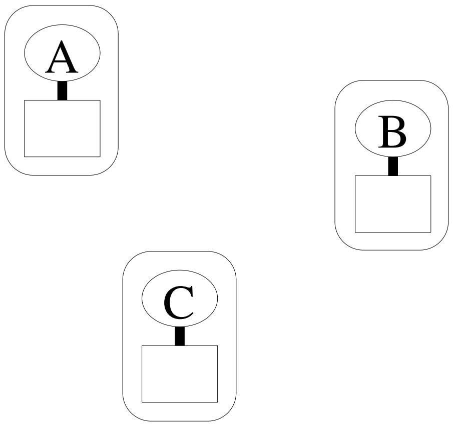
Pipeline-Architekturen sind im UNIX-Bereich weit verbreitet. So können beispielsweise verschiedene Werkzeuge zur Erstellung und Manipulation von Dokumenten und Programmen verwendet werden (z.B. *sed* und *awk*), die Ergebnisse werden an weitere Werkzeuge, wie das Satzsystem *troff* oder an Präprozessoren und nachfolgende Übersetzer von Programmiersprachen weitergereicht.

2.2.4.3 Residente Architekturen

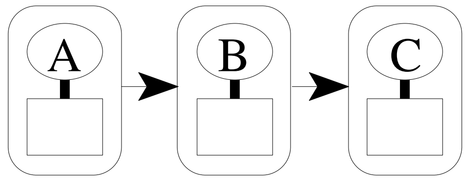
Residente Architekturen bieten einen ersten Ansatz zur Integration von Werkzeugen. Abbildung 1 c) gibt den schematischen Aufbau wieder. Hier werden die Repräsentationen aller Werkzeuge in einem gemeinsamen Adreßraum gehalten. Somit ist die Unterstützung bei dem Austausch von Informationen durch die Architektur gegeben, es können beliebige Mechanismen zum Informationstransfer implementiert werden. Bisherige Implementierungen nach dieser Architektur, so z.B. die Interlisp-Umgebung [TM81] und Smalltalk-Umgebungen [Gold84], leiden unter komplexen Abhängigkeiten zwischen den Werkzeugen, alle Werkzeuge müssen zugleich aktiv sein. Daraus resultiert eine gewisse Schwerfälligkeit und große Anforderungen an die zugrunde liegende Plattform.

2.2.4.4 Busarchitekturen

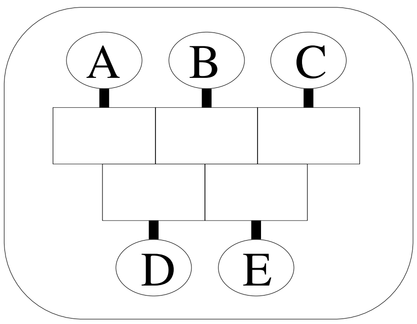
Busarchitekturen bieten die Möglichkeit, Werkzeuge mittels eines Kommunikationskanals zu entkoppeln. Dies wird in Abbildung 1 d) dargestellt. Die klare Trennung der Werkzeuge durch die Definition eindeutiger Kommunikationsschnittstellen führt zu der Möglichkeit einzelne Werkzeuge austauschen zu können. Die Komplexität ist im Vergleich zu einer residenten Architektur geringer, da nicht alle Werkzeuge gleichzeitig aktiv sein müssen. Das Hauptproblem von Busarchitekturen ist in der Praxis die unausgewogene Verteilung von Funktionalität zwischen den Werkzeugen, so daß der Kommunikationskanal zum Flaschenhals werden kann. Beispiele für Busarchitekturen sind der ESF-Softwarebus [GHHM+92] und PCTE (*portable common tool environment*) [WJ93].



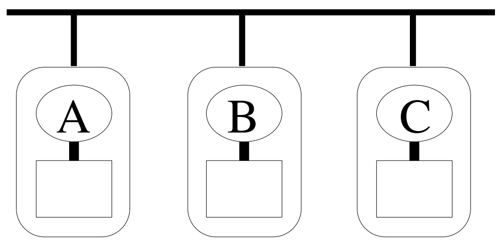
a) Separate Werkzeuge



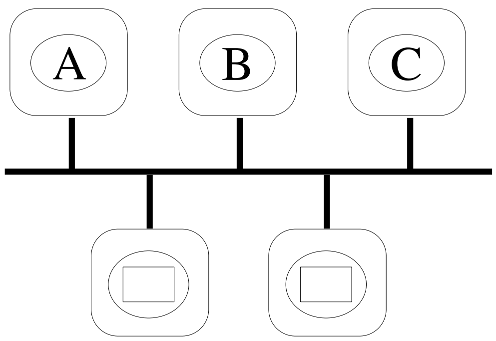
b) Pipeline-Architektur



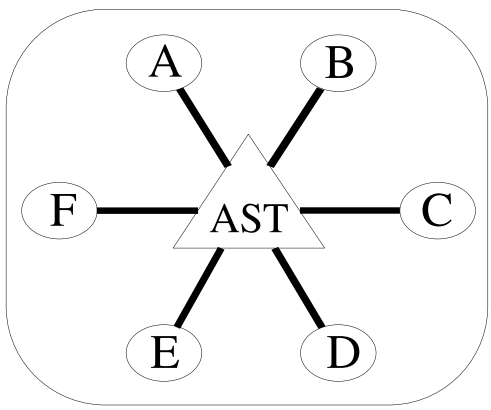
c) Residente Architektur



d) Busarchitektur



e) Client/Server-Architektur



f) Grammatikbasierte Architektur

Abbildung 1 Verschiedene Architekturarten für Software-Entwicklungsumgebungen

2.2.4.5 Client/Server-Architekturen

Die Client/Server-Architekturen sind ein bestimmter Anwendungsfall der Busarchitektur. Abbildung 1 e) stellt den schematischen Aufbau dar. Die Repräsentationen werden in dedizierte Server ausgelagert, auf diese Repräsentationen greift die Werkzeugfunktionalität über den Kommunikationskanal zu. Client/Server-Architekturen bilden einen ersten Ansatz die mehrfache Haltung von Informationen zu vermeiden, jedoch kann in Client/Server-Architekturen (wie in Busarchitekturen) die Kommunikation zum Flaschenhals werden.

2.2.4.6 Grammatikbasierte Architekturen

Auch grammatikbasierte Architekturen bieten architekturseitig Unterstützung bei der Vermeidung der doppelten Haltung von Informationen. Die Werkzeuge greifen auf die gemeinsame Informationsstruktur zurück. Abbildung 1 f) zeigt den Aufbau der grammatikbasierten Architektur. Grammatikbasierte Architekturen zeichnen sich durch die Benutzung von attributierten abstrakten Syntaxbäumen als zentraler Informationsstruktur für die Handhabung von Dokumenten im Hinblick auf die Verwendung in Software-Entwicklungsumgebungen aus. Diese zentrale Baumstruktur ist als mit AST beschriftetes Dreieck in der Abbildung dargestellt.

Attributierte abstrakte Syntaxbäume dienen als Integrationsbasis zwischen allen bei der Software-Entwicklung anfallenden Dokumenten. Sie realisieren eine Abbildung der Dokument- und Inkrementstrukturen auf Baumstrukturen. Durch Attributierung der Baumstruktur wird die Verwaltung zusätzlicher Informationen und eine symmetrische Behandlung betreffend Annotation und Querverweisen ermöglicht. Die Repräsentation der Dokumente erfolgt durch die Baumstruktur und gemeinsam genutzte Attribute, weitere Attribute können werkzeugspezifische Informationen beinhalten.

Grammatikbasierte Architekturen sind bisher fast ausschließlich universitären Ursprungs. Als Beispiele seien Gandalf [HN86], Mentor [DKLM84], Synthesizer Generator [RT87], Mjølner BETA und Orm [KLLM94] genannt. IPSEN [ES89] verwendet eine Grammatikbasierung in Form von Syntaxgraphen als zentrale Informationsstruktur.

Die Aufgabe dieser Diplomarbeit besteht in der Verwendung einer grammatikbasierten Architektur für die Entwicklung einer Prototyping-Umgebung.

2.3 ProSet

Im Mittelpunkt der zu erstellenden Prototyping-Umgebung steht die *very high level*-Sprache ProSet. ProSet (das Akronym für *PROtotyping with SETs*) ist eine mengentheoretisch orientierte imperative Programmiersprache, die für eine schnelle Konstruktion von ausführbaren Prototypen geeignet ist. ProSet wurde als Nachfolger von SETL [DF89] an der Universität Essen entworfen und implementiert und später an der Universität Dortmund weiterentwickelt. Im folgenden wird ein kurzer Überblick über die Spracheigenschaften gegeben, soweit dies für das Verständnis dieser Arbeit von Interesse ist. Eine ausführlichere Beschreibung findet der interessierte Leser in der Sprachdefinition [DFG+92a]. Abschließend wird der aktuelle Stand der Entwicklung von ProSet diskutiert. Dies umfaßt die Implementierung der Sprache sowie die bisher existierenden Werkzeuge zur Programmentwicklung.

2.3.1 Spracheigenschaften

In den folgenden Abschnitten werden besondere Eigenschaften der Sprache PROSET erläutert. Anschließend an die Diskussion der Datentypen, Kontrollstrukturen, Prozeduren und des Modulkonzeptes werden das Persistenzkonzept, das Ausnahmebehandlungskonzept, sowie die Möglichkeiten der Entwicklung paralleler Programme vorgestellt. Nach der Diskussion über PROSET als Breitbandsprache bilden ein Blick auf das Typsystem und die ausführliche Beschreibung der Sichtbarkeitsregeln den Abschluß. Das Typsystem und die Sichtbarkeitsregeln sind für die statische Analyse von PROSET-Programmen von besonderer Bedeutung.

2.3.1.1 Datentypen

Neben den Basisdatentypen `integer`, `real`, `boolean`, `string` und `atom` bietet PROSET insbesondere die Datentypen aus der endlichen Mengenlehre: `set` (Mengen) und `tuple` (Tupel). Mittels dieser Typen können dann beispielsweise Abbildungen und Relationen einfach ausgedrückt werden. Für die Beschreibung von Mengen und Tupeln stehen die gewohnte mathematische Notation und die üblichen Operationen zur Verfügung. Mengen und Tupel müssen nicht homogen sein, d.h. sie können aus beliebigen PROSET-Werten zusammengesetzt sein. Der undefinierte Wert `om` bildet hierbei eine Ausnahme, er darf nicht Element einer Menge sein. Zusätzlich bietet PROSET die höheren Datentypen `function`, `modtype` und `instance`. Sie beschreiben Funktionen und Module und werden an den entsprechenden Stellen in den folgenden Abschnitten näher beschrieben.

Alle Werte der genannten Typen besitzen Bürgerrechte erster Klasse, sie besitzen eine Identität und können zugewiesen werden, d.h. sie können als Elemente von Mengen oder Tupeln oder als Parameter von Prozeduren und Funktionen verwendet werden.

2.3.1.2 Kontrollstrukturen

PROSET bietet ähnliche Kontrollstrukturen wie andere von Algol abstammende Sprachen. Dies sind die bedingten Anweisungen und die üblichen Schleifenkonstrukte. Letztere sind an die endliche Mengenlehre angepaßt, um beispielsweise die Iteration über zusammengesetzte Datentypen zu ermöglichen.

2.3.1.3 Programmstruktur

PROSET ist eine blockstrukturierte Sprache. Ein Programm besteht aus dem Hauptprogramm innerhalb dessen weitere Programmeinheiten wie benannte und anonyme Prozeduren, Module und Behandlungsroutinen (*handler*) definiert werden können. Innerhalb dieser Einheiten können wiederum weitere Programmeinheiten geschachtelt sein.

2.3.1.4 Prozeduren

PROSET bietet als Strukturierungsmittel unter anderem benannte Prozeduren und anonyme Prozeduren (*lambda*) an. Diese Prozeduren können parametrisiert sein und Werte als Ergebnis liefern. Die Parameterübergabe kann auf drei Arten erfolgen, die den bekannten Mechanismen *call-*

by-value, *call-by-result* und *call-by-value/result* entsprechen. Dementsprechend werden die formalen Parameter mit `rd`, `wr` und `rw` gekennzeichnet.

Als `wr`-Parameter und `rw`-Parameter dürfen nur gültige l-Werte übergeben werden. Diese bezeichnen Ausdrücke, die auf der linken Seite einer Zuweisung vorkommen dürfen. Im folgenden werden die syntaktischen Konstrukte aufgeführt, mit denen gültige l-Werte für PROSET geformt werden können:

- Bezeichner sind l-Werte, außer wenn sie Konstanten, Prozeduren, Module oder Behandlungsroutinen bezeichnen
- Tupel- oder Zeichenkettenselektionen und Ausschnittsbildung (*slicing*) durch $l(e)$, $l(e..)$ und $l(e_1..e_2)$, wobei l ein l-Wert ist. Zusätzliche semantische Einschränkungen für die Ausdrücke e, e_1 und e_2 sollen in diesem Zusammenhang nicht berücksichtigt werden.
- $l(e_1, ..., e_n)$ und $l\{e_1, ..., e_n\}$ wobei l ein l-Wert ist. Hierbei handelt es sich um assoziativen Zugriff auf Abbildungen.
- Zusammengesetzte l-Werte $[l_1, ..., l_n]$, wobei l_1 bis l_n l-Werte oder das Auslassungssymbol – sind. Zusammengesetzte l-Werte, die das Auslassungssymbol enthalten, dürfen nicht als `rw`-Parameter verwendet werden, da sie selbst keinen gültigen Wert besitzen.

Aufgrund der schwachen Typisierung von PROSET (siehe Abschnitt 2.3.1.10) sind die Parameter von Prozeduren und anonymen Prozeduren polymorph.

Prozeduren können Seiteneffekte verursachen, indem sie nicht-lokale Objekte verwenden (siehe Abschnitt 2.3.1.11). Funktionen werden durch Einfrieren der Werte von nicht-lokalen Objekten erzeugt. Durch die Anwendung des `closure`-Konstruktes auf einen Prozedurbezeichner wird ein Wert des Typs `function` erzeugt.

2.3.1.5 Modulkonzept

Zur Unterstützung des Programmierens im Großen bietet PROSET Module und Modulinstanzen an. Module sind dabei als Menge von Prozeduren zu verstehen, die auf einer gemeinsamen Datenbasis arbeiten. Der Zugriff auf diese Datenbasis kann ausschließlich über exportierte Prozeduren (Modulprozeduren) erfolgen. Das `closure`-Konstrukt ist auf Module anwendbar und liefert als Ergebnis einen Wert mit Bürgerrechten erster Klasse vom Typ `modtype`. Aus einem derartigen Wert erhält man durch Instantiierung einen Wert vom Typ `instance`, der wiederum Bürgerrechte erster Klasse besitzt.

2.3.1.6 Persistenzkonzept

In PROSET können alle Werte mit Bürgerrechten erster Klasse persistent gemacht werden, so daß sie die Programmausführung überdauern, in der sie erzeugt worden sind. Dazu muß für persistente Werte die Deklaration eines Bezeichners mit `persistent` erfolgen. Persistente Werte können von demselben Programm in einem anderen Lauf oder auch von anderen Programmen verwendet werden. Durch die persistente Speicherung der Programmeinheiten Funktion, Modul und Instanz wird getrennte Übersetzbarkeit und Wiederverwendbarkeit erreicht. Ebenso ist auch die Kommunikation von Programmen durch persistente Werte möglich.

Die Speicherung der persistenten Werte erfolgt innerhalb von Datencontainern, die im PROSET-Sprachgebrauch als *p-files* bezeichnet werden. Diese *p-files* enthalten in der Regel neben den gespeicherten Werten auch ein Inhaltsverzeichnis der enthaltenen benannten Werte.

2.3.1.7 Ausnahmebehandlung

Der Mechanismus zur Behandlung von Ausnahmesituationen bietet die Möglichkeit, sowohl Fehlersituationen als auch andere anormale Situationen getrennt vom eigentlichen Kontrollfluß eines Algorithmus zu behandeln. Hervorzuheben ist die Unterscheidung von Ausnahme (*exception*) und Behandlungroutine (*handler*), zwischen denen dynamisch Assoziationen hergestellt werden. ProSet bietet bei der Ausnahmebehandlung die Möglichkeit zum Abbruch oder zur Wiederaufnahme der ausnahmeauslösenden Programmeinheit.

2.3.1.8 Entwicklung paralleler Programme

Für die Entwicklung paralleler Programme verfügt ProSet, neben der Möglichkeit der Prozeßgenerierung, über einen Synchronisations- und Kommunikationsmechanismus auf der Basis von Tupelräumen. Diese bilden einen virtuellen, gemeinsamen Datenbereich für Prozesse. Das Auslesen von Tupeln aus einem Tupelraum ist assoziativ, es werden zu einem angegebenen Muster passende Tupel ausgelesen. Durch die Verwendung des Tupelraumkonzeptes wird von der zugrunde liegenden parallelen oder verteilten Architektur abstrahiert.

2.3.1.9 Breitbandsprache

ProSet erlaubt die Umsetzung mengentheoretisch formulierter Algorithmen auf einem sehr hohen Niveau. Die Lesart eines ProSet-Programms muß sich dabei kaum von der eines Algorithmus unterscheiden. Zugleich ist auch eine Formulierung von Problemen auf weniger abstrakten Niveaus möglich. Beispielsweise können Iterationen oder Quantifizierungen über Mengen sowohl mathematisch notiert als auch in Form expliziter Schleifen formuliert werden. Dies ermöglicht die Transformation mengentheoretisch formulierter Prototypen in effiziente, semantisch äquivalente Programme.

2.3.1.10 Typisierung

ProSet ist eine schwach typisierte Sprache. Die in ProSet-Programmen verwendeten Variablen müssen nicht explizit deklariert werden. Zudem kann der Typ einer Variablen zur Laufzeit wechseln. Die schwache Typisierung wirkt sich auf den Zeitpunkt der Überprüfbarkeit von Fehlern aus: Typüberprüfungen können erst zur Laufzeit durchgeführt werden. Zusätzlich sind alle nicht explizit deklarierten Bezeichner grundsätzlich implizit deklariert. Tippfehler bei Bezeichnern oder ähnliche Fehler können somit nicht statisch erkannt werden und führen in vielen Fällen zu einem unerwarteten Verhalten des Prototypen.

Abbildung 2 zeigt ein Beispielprogramm, in dem die als `visible` deklarierte Variable `x` den Typ wechselt. Die Variable `y` wird implizit deklariert.

```
program ...
   visible
      x:=5;      -- x ist vom Typ integer
begin
   x := 5.5;     -- x ist nun vom Typ real
   ...
   y := 0;       -- y ist nicht explizit deklariert
end ...
```

Abbildung 2 ProSet: Beispiel zur Typisierung

```
1   program Bereiche;
2      visible a;   -- definierende Auftreten von
3      hidden b;    -- a und b
4   begin
5      p();         -- angewendetes Auftreten von p
6      procedure p();
7         visible c;
8      begin
9         c:=a;     -- Anwendung von a aus Zeile 2
10        c:=b;     -- implizite Deklaration von b, da b in
11                  -- Zeile 2 als hidden deklariert wurde
12     end p;
13  end Bereiche;
```

Abbildung 3 PROSET: Bereiche, globale und lokale Deklarationen

2.3.1.11 Sichtbarkeitsregeln

Besondere Bedeutung kommt im Rahmen dieser Arbeit der Auswertung von Sichtbarkeitsregeln bei der statischen Analyse von PROSET-Programmen zu. Aus diesem Grund werden in diesem Abschnitt ausführlich die Sichtbarkeitsregeln für Bezeichner in PROSET vorgestellt.

Bereiche und Deklarationsstellen

PROSET stellt eine Reihe von Sprachkonstrukten zur Verfügung, in denen Deklarationen von Bezeichnern vorgenommen werden können. Dies sind die Konstrukte Hauptprogramm, Prozedur, Lambda, Modul und Behandlungsroutine. Diese Konstrukte bilden einen **Bereich** (*range*), in dem die deklarierten Namen dieses Konstruktes sichtbar sind. Dabei sind Bereiche nicht Teil des umgebenden Bereiches. Der Begriff des Bereiches ist für die nachfolgende Definition der Sichtbarkeit von Bedeutung. Abbildung 3 zeigt ein Programmbeispiel mit Zeilennumerierung. Dieses Programm enthält zwei Bereiche. Der zu der Programmdefinition korrespondierende Bereich umfaßt die Zeilen 2 bis 6 (ohne die formale Parameterliste) und die Zeile 13. Der zweite Bereich korrespondiert zur der Prozedur `p` und umfaßt die Zeilen 6 (beginnend mit der formalen Parameterliste) bis 12.

In Deklarationen werden Bezeichner mit Objekten assoziiert. Das Vorkommen der Bezeichner in diesem Kontext wird als **definierendes Auftreten** (auch Namensdeklaration) bezeichnet. Die Verwendung eines Bezeichners zur Referenzierung des assoziierten Objektes wird als **angewendetes Auftreten** (auch Namensanwendung) bezeichnet. Jedes angewendete Auftreten bezieht sich in PROSET auf höchstens ein definierendes Auftreten. Die Kommentare im Programm in Abbildung 3 weisen auf definierende und angewendete Auftreten von Bezeichnern hin.

Lokale oder globale Sichtbarkeit

PROSET gestattet die Beeinflussung der Sichtbarkeit von explizit deklarierten Variablen und Konstanten durch die Verwendung der Deklarationsattribute `hidden` und `visible`. Eine Deklaration, die mit dem Attribut `hidden` gekennzeichnet ist, beschränkt die Sichtbarkeit eines Bezeichners auf den umgebenden Bereich und wird als lokale Deklaration bezeichnet. Eine Deklaration mit dem Attribut `visible` wird als globale Deklaration bezeichnet. Sie erweitert die Sichtbarkeit des Bezeichners auf die den aktuellen Bereich enthaltenen Bereiche. Globale

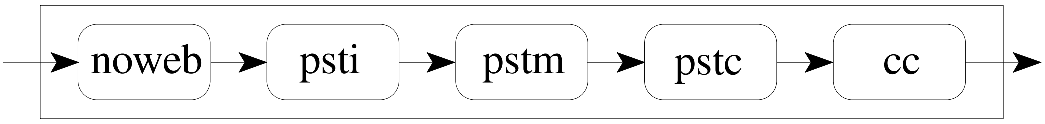


Abbildung 4 Pipeline-Architektur des PROSET-Übersetzers

Deklarationen können in inneren Bereichen überdeckt werden. Ist für Variablen oder Konstanten keines der Attribute angegeben, so ist der Bezeichner lokal sichtbar. Die Sichtbarkeit der Namen von Prozeduren, Modulen, Behandlungsroutinen und formalen Parametern kann nicht beeinflußt werden, diese sind immer global sichtbar. Abbildung 3 zeigt Beispiele für lokale und globale Deklarationen.

Die in PROSET verwendeten Iteratoren für Mengen und Tupel führen implizite Deklarationen ein. Die implizit deklarierten Bezeichner sind innerhalb des Konstrukts sichtbar, in dem der Iterator verwendet wird: dies sind Schleifen mit `whilefound` und `for`, quantifizierte Ausdrücke mit `exists` und `forall`, sowie Mengen- und Tupelformungsausdrücke mit Iterator. Diese Konstrukte bilden selbst keinen Bereich, sondern führen ausschließlich einen temporären, nur in diesem Konstrukt gültigen Namen ein.

Namensräume

PROSET verwendet drei Namensräume. Es gibt einen Namensraum für die Namen des Programms, der Prozeduren, Module und Behandlungsroutinen, der Variablen, der Konstanten und der formalen Parameter. Ein weiterer Namensraum existiert für Schleifenmarken (*labels*). Der dritte Namensraum ist flach und existiert im Rahmen von Ausnahmebezeichnern.

2.3.2 Stand der Entwicklung von PROSET

Der Sprachentwurf von PROSET ist inzwischen weitestgehend abgeschlossen. PROSET wurde im Rahmen einer Projektgruppe an der Universität Dortmund in größerem Rahmen eingesetzt und evaluiert [PG95]. Neben einem Übersetzer für PROSET-Programme in die Sprache C und einem einfachen Werkzeug zur Fehlersuche (*debugger*) existieren noch weitere Werkzeuge, wie das *p-file tool* zur Erzeugung von *p-files*, der *p-file editor*, ein Werkzeug zur Analyse und Verwaltung persistenter PROSET-Werte [Kapp95] und ein Werkzeug zur Fehlersuche in parallelen PROSET-Programmen [Pohl96]. Weitere Spracherweiterungen und Werkzeuge werden zur Zeit im Rahmen von Diplomarbeiten und Dissertationen erarbeitet.

Als Plattform für alle Werkzeuge dienen Sun Sparc Rechner mit den Betriebssystemen Solaris 1 und 2. Als Basis für den Persistenzmechanismus dient eine hochperformante Implementierung des PCTE-Standards [WJ93].

Zur Erstellung von PROSET-Programmen in Form von *noweb*-Dokumenten mit Dokumentation und Quelltext werden momentan traditionelle Texteditoren verwendet. Die zu übersetzenden Dokumente werden als Eingabe einer Pipeline verschiedener Werkzeuge übergeben (beteiligt sind z.B. Quelltextextraktion, Inkludierungsverarbeitung und Makroersetzung, der eigentliche Übersetzer von PROSET nach C, die nachfolgende Übersetzung in Objektdateien und das Binden). Als Ausgabe der Pipeline erhält man ausführbare Programme. Jedes in dieser Pipeline ausgeführte Werkzeug operiert auf einer eigenen internen Repräsentation des Programms mit unterschiedlichen

Datenstrukturen. Die Kommunikation erfolgt mittels Textdateien. Abbildung 4 zeigt schematisch und vereinfacht den Ablauf der Übersetzung eines PROSET-Programms mit Quelltextextraktion, Inkludierungsverarbeitung, Makroersetzung, Übersetzung nach C und in Objektdateien.

2.4 Grammatikbasierte Entwicklungsumgebungen

Aufgabe dieser Diplomarbeit ist die Entwicklung einer grammatikbasierten Prototyping-Umgebung für die Prototyping-Sprache PROSET. In den folgenden Abschnitten werden Techniken und Werkzeuge diskutiert, die für die Erstellung grammatikbasierter Umgebungen von zentraler Bedeutung sind. Dazu werden Begriffsbestimmungen für attributierte abstrakte Syntaxbäume als zentrales Repräsentationsmittel von Dokumenten und für Spezifikationstechniken dieser attributierten abstrakten Syntaxbäume vorgenommen. Anschließend werden die Begriffe syntaxbasierter Editor und Unparser als zentrale Werkzeuge grammatikbasierter Umgebungen eingeführt. Die Betrachtung des BETA-Metaprogrammiersystems, das für die Entwicklung der PU zu verwendende Basissystem, bildet den Abschluß.

2.4.1 Attributierte abstrakte Syntaxbäume

Der **abstrakte Syntaxbaum** (AST) bildet eine für die Darstellung von Programmen inzwischen allgemein anerkannte Datenstruktur. Abstrakte Syntaxbäume wurden zuerst für Lisp-Programme [McCa62] und später stärker formalisiert im Mentor-Projekt [DKLM84] für Pascal-Programme eingesetzt.

Abstrakte Syntaxbäume stellen strukturelle Zusammenhänge von (syntaktisch) korrekten Programmen dar, sie abstrahieren von einer konkreten Darstellung eines Programms. ASTs sind knotenmarkierte Bäume. Die Semantik der Beziehung zwischen einem Knoten und dessen Sohnknoten ist eine "besteht aus" oder "ist zusammengesetzt aus"-Beziehung. ASTs bilden somit die Komposition von Programmen auf eine Baumstruktur ab. Die Knoten des Baumes sind jeweils mit dem Namen eines syntaktischen Konstrukts der Sprache markiert. Bezeichner und Literale werden üblicherweise als atomare Blattknoten verwaltet. Auf eine weitergehende formale Definition wird in diesem Rahmen verzichtet. Bisher ist keine allgemein anerkannte formale Definition bekannt.

ASTs dienen nicht nur der Repräsentation vollständiger Programme, durch Ersetzen von Teilbäumen durch Platzhalter erhält man **unvollständige Programme**. Die Gestalt der ASTs für eine Programmiersprache wird spezifiziert durch Angabe einer Grammatik für wohlgeformte Bäume (**Baumgrammatik**), dies wird im folgenden Abschnitt näher beschrieben. Da bei der Spezifikation gewisse Gestaltungsfreiheiten gegeben sind, ist auch die Darstellung eines Programms abhängig von der Spezifikation. Für den Austausch von ASTs zwischen verschiedenen SEUs ist die Festlegung auf eine bestimmte formale Spezifikation unerläßlich. Dies ist für die Sprache Ada in Form einer Beschreibung in DIANA [GH83] durchgeführt worden.

Abbildung 5 zeigt ein unvollständiges PROSET-Fragment und ein Beispiel für einen zugehörigen abstrakten Syntaxbaum. Platzhalter sind in diesem Beispiel durch Namen in doppelten spitzen Klammern dargestellt. Das Programm besteht aus einer Prozedurdefinition, die eine explizite Deklaration und zwei Anweisungen enthält, den Aufruf einer Prozedur und eine Zuweisung. Die Deklaration und die Anweisungen enthalten mehrere Bezeichner. Der AST stellt diese Bestandteile und ihre Komposition dar: Der Knoten *ProcDef* bezeichnet die Prozedurdefinition und bildet den

```
procedure <<Name>> ();
   visible a;
begin
   q ();
   b := a;
end <<Name>>;
```

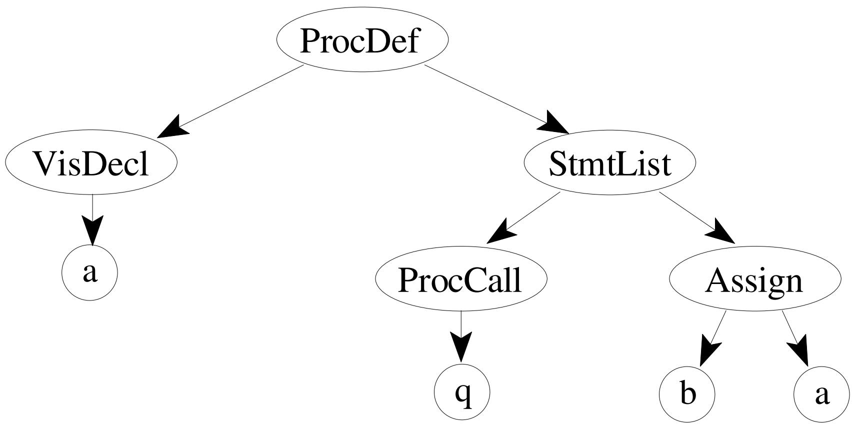


Abbildung 5 Abstrakter Syntaxbaum (Beispiel)

Wurzelknoten des Baumes. Die Bestandteile werden als die Sohnknoten *VisDecl*, *ProcCall* und *Assign* dargestellt, wobei für die Verwaltung der Liste der Anweisungen der zusätzliche Knoten *StmtList* enthalten ist. Die Komponenten werden durch gerichtete Kanten verknüpft. Die Blätter werden von Lexemknoten mit dem repräsentierten Namen als Beschriftung gebildet. Innerhalb dieses AST sind die unvollständigen Stellen durch leere Teilbäume dargestellt.

Erst durch Erweiterungen wird die Datenstruktur AST ausreichend mächtig, um als zentrale Repräsentation einer SEU Verwendung zu finden. Beispielsweise ist es oftmals nötig logische Zusammenhänge zwischen weit im Baum entfernten Knoten darstellen zu können.

Die Erweiterung durch Attributierung der Baumknoten ist als Weg vorgeschlagen worden, um eine ausreichend mächtige Repräsentation für Dokumente zu erhalten. **Attribute** beinhalten weitere Informationen zu einem Knoten, wie z.B. Kommentare, logische Zusicherungen, generierten Maschinencode, Informationen für das Unparsen (siehe Abschnitt 2.4.4) oder für die Analyse der statischen Semantik. Diese Aufzählung zeigt, daß Attribute durchaus eine komplexe Struktur besitzen können.

2.4.2 Spezifikation attributierter abstrakter Syntaxbäume

Für die Verwendung als zentrale Informationsstruktur ist eine formale Spezifikation gültiger attributierter abstrakter Syntaxbäume unerläßlich. Die formale Spezifikation wird in den drei Bereichen abstrakte Syntax, konkrete Syntax und Attributierung vorgenommen.

Als Spezifikation wird eine aus der kontextfreien Grammatik einer Sprache entwickelte **abstrakte Grammatik** als Baumgrammatik verwendet, welche die Gestalt der zu repräsentierenden Bäume festlegt. Jede Produktion der Baumgrammatik beschreibt die Struktur einer Knotenart durch Definition von Anzahl und Typen der Sohnknoten. Die Knoten eines AST sind dann entsprechend der zugehörigen Produktion der Baumgrammatik markiert.

Die verbreiteten Notationen für die Spezifikation der abstrakten Syntax basieren auf den Ausdrucksmitteln **Aggregation** und **syntaktischen Typen**. Die syntaktischen Typen bilden Hierarchien und werden in den Aggregationen verwendet, um dort die Typen der Sohnknoten zu beschreiben. Eine bekannte Notation für die Spezifikation der abstrakten Syntax ist die z.B. im Synthesizer Generator [RT87] verwendete Beschreibung mit Phyla und Operatoren (vergleiche auch [ES89]). Eine weitere auf Aggregationen und syntaktischen Typen basierende Notation ist GRAMPS, das in Abschnitt 2.4.5.2 genauer beschrieben wird.

Die **konkrete Grammatik** beschreibt die Gewinnung einer für den Benutzer lesbaren Darstellung aus dem AST. Die konkrete Grammatik enthält dazu ausschließlich Präsentationsaspekte, für die verschiedenen Arten von Baumknoten wird deren Aussehen spezifiziert. Die Präsentation kann graphisch, textuell oder als Kombination erfolgen. Im Fall der textuellen Präsentation wird das Aussehen der Baumknoten üblicherweise anhand von Unparsing-Schemata beschrieben (vergleiche Abschnitt 2.4.4).

Die Spezifikation von **Attributwerten** kann einerseits operational durchgeführt werden. Beispielsweise wurde der Ansatz der Aktionsprozeduren (*action routines*) aus dem Übersetzerbau auf den Bereich Entwicklungsumgebungen übertragen [ES89, KLLM94]. Hierbei wird jeder Produktionen der Baumgrammatik eine Aktionsprozedur zugeordnet, diese Prozedur wird bei Einfügungen, Löschungen oder Überprüfungen aufgerufen.

Andererseits wurden ebenso deklarative Ansätze, wie z.B. attributierte Grammatiken auf Entwicklungsumgebungen übertragen. Attributierte Grammatiken beschreiben die Berechnung von Attributwerten eines Knotens aus den Attributwerten von Vater- oder Sohnknoten in Form von Attributgleichungen [RT87].

2.4.3 Syntaxbasierte Editoren

Innerhalb von grammatikbasierten Entwicklungsumgebungen ist es sinnvoll, die Programmerstellung mittels Editoren neu zu überdenken. Traditionell werden zur Programmerstellung Texteditoren verwendet, mit denen Programmtexte zeichenweise erstellt werden. Dieser Ansatz bleibt allerdings hinter den Möglichkeiten zurück, die in grammatikbasierten Umgebungen (durch die Kenntnis der Sprachsyntax und der Darstellung von Programmen als abstrakte Syntaxbäume) realisierbar sind.

Syntaxbasierte Editoren bieten eine Alternative zu der traditionellen textbasierten Erstellung von Programmen. Statt Programme in Form von Zeichenfolgen zu editieren, wird in syntaxbasierten Editoren ein Programm auf der Basis von Inkrementen bearbeitet. Der Editor muß Kenntnisse über die zugrundeliegende Programmiersprache haben. Die Programmentwicklung erfolgt üblicherweise durch die Ersetzung von Platzhaltern durch Programmkonstrukte, die als Schablone aus Schlüsselwörtern und weiteren Platzhaltern in das Programm eingefügt werden. Die Platzhalter der Schablonen werden sukzessive durch Ausfüllen (Auswahl weiterer Sprachkonstrukte) ersetzt, bis ein vollständiges Programm entsteht, das keine weiteren Platzhalter enthält. Eine wesentliche, vorteilhafte Eigenschaft syntaxbasierter Editoren ist die Erstellung syntaktisch korrekter Programme. Weitere Vor- und Nachteile syntaxbasierter Eingabe werden in Abschnitt 3.3.3 erläutert.

Hybrideditoren vereinigen die Möglichkeiten der syntaxbasierten und der freien textuellen Editierung. Sie bieten sowohl syntaxbasierte Eingabe mit Unterstützung bei der Erstellung von syntaktisch korrekten Programmen, als auch freie Eingabe in Form von Programmtext ohne syntaktische Unterstützung. Nach dem Abschluß der freien Eingabe muß der erstellte Programmtext auf syntaktische Korrektheit überprüft werden. Als **freie Eingabe** bezeichnet man die Editierung eines Teilbaumes des Programms in textueller Form und die anschließende Überführung in einen Teil-AST für dieses Programmteil, welcher dann wieder in den AST des Programms eingefügt wird.

2.4.4 Unparser

Die Repräsentation von Programmen als abstrakte Syntaxbäume führt zu der Notwendigkeit einen Mechanismus zu finden, der die interne Repräsentation in eine für den Menschen lesbare und adäquate Darstellung transformiert. Der Vorgang der Erzeugung einer solchen Darstellung wird **unparsen** (*unparsing*) genannt, das zugehörige Werkzeug wird im folgenden als **Unparser** bezeichnet.

Die Darstellung von Programmen kann einerseits textuell, andererseits auch graphisch erfolgen. Bei textuellen Unparsern wird üblicherweise die gewohnte Notation als Quelltext erzeugt, die abstrakte Repräsentation wird im wesentlichen um Schlüsselworte erweitert. Dies erreicht man im einfachsten Fall durch die Zuordnung von Schemata zu den Knoten des abstrakten Syntaxbaumes. Zusätzlich gibt es weitere Eigenschaften des Unparsers, die das Erscheinungsbild des dargestellten Programms beeinflussen. Diese Eigenschaften sind größtenteils orthogonal und können kombiniert werden:

Kontextsensitive Darstellung: Die Darstellung eines Knotens wird durch den aktuellen Kontext beeinflußt.

Bedingte Darstellung: Knoten werden in Abhängigkeit von einer globalen Bedingung unterschiedlich präsentiert.

Adaptive Darstellung: Die Darstellung paßt sich an verfügbaren Ausgabebereich, z.B. an die zur Verfügung stehende Breite an.

Auslassung (*elision*): Teile des Programms werden nicht dargestellt, sondern durch ein Auslassungszeichen kenntlich gemacht. Mittels Auslassungen können Programme in verschiedenen Detaillierungsgraden dargestellt werden.

Mehrfachsichten: Programme können auf verschiedene Arten dargestellt werden.

2.4.5 Das BETA-Metaprogrammiersystem

Im Rahmen der Erforschung von objektorientierten Software-Entwicklungsumgebungen im Mjølner-Projekt [KLLM94], das in Dänemark und Skandinavien unter Beteiligung mehrerer Universitäten, Forschungseinrichtungen und Firmen durchgeführt wurde, entstand das **Mjølner BETA-System**, eine grammatikbasierte Entwicklungsumgebung für die objektorientierte Programmiersprache BETA [MMPN93, Mjøl94c].

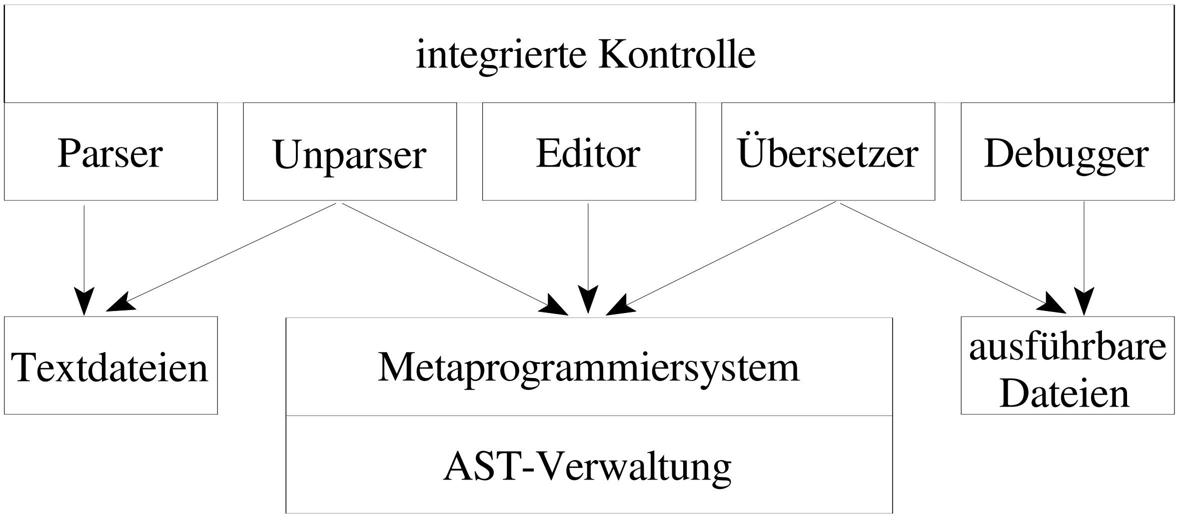


Abbildung 6 Mjølner BETA-System (schematische Darstellung)

Abbildung 6 zeigt den schematischen Aufbau des Mjølner BETA-Systems. Es besteht aus den Werkzeugen syntaxbasierter Editor, Parser, Unparser, Übersetzer und weiteren zum Teil BETA-spezifischen Werkzeugen. Alle Werkzeuge basieren auf einer Komponente zur Repräsentation von Programmen als AST.

Für die Entwicklung einer Prototyping-Umgebung im Rahmen dieser Diplomarbeit hat die Firma Mjølner-Informatics eine Teilmenge der Entwicklungsumgebung als Rahmenwerk (*application framework*) zur Verfügung gestellt. Das Rahmenwerk umfaßt den Editor, Parser und Unparser und wurde als frühe Vorabversion der Version 5.0 zusammen mit der Version 3.1 des BETA-Systems bereitgestellt. Dieses Rahmenwerk kann im Bereich der Benutzerschnittstelle durch das Hinzufügen von Dialogen und Menüs und die Integration zusätzlicher Werkzeuge erweitert werden.

Mit dem Übergang von der Version 4.9 der BETA-Entwicklungsumgebung (in den Versionen 3.0 und 3.1 des BETA-Systems) zu der Version 5.0 (die zukünftig mit der Version 4.0 des BETA-Systems verfügbar sein wird) gehen umfassende Änderungen in der Benutzerschnittstelle einher. Während die Umgebung der Version 4.9 auf den Motif-Bibliotheken basierte, ist die Oberfläche der Version 5.0 durch eine plattformunabhängige Zwischenschicht (*guiEnv* [Mjøl95]) realisiert worden. Diese Zwischenschicht bildet zur Zeit die Funktionalität wiederum auf die Motif-Bibliotheken ab, mit Hilfe dieser Zwischenschicht ist aber erstmals die Erweiterbarkeit, wie sie in [Mjøl94b] vorgestellt wurde, in Form eines Rahmenwerks realisiert worden.

Die Basis für die Repräsentation von Programmen als AST bildet das **BETA-Metaprogrammier-system** (MPS), das die persistente Speicherung und den Zugriff auf ASTs in Form einer in BETA realisierten Bibliothek zur Verfügung stellt. Das MPS bildet sowohl die zentrale Komponente des Rahmenwerks für die Realisierung von Entwicklungsumgebungen, als auch die Erstellung weiterer grammatikbasierter, sprachspezifischer Werkzeuge. Neben der Handhabung von ASTs bietet das MPS weitere Werkzeuge zur Unterstützung der Erstellung von Entwicklungsumgebungen, so z.B. einen Parser-Generator zur Erzeugung tabellengetriebener Parser, einen Generator für die Erzeugung von Unparsing-Schemata und einen Generator für die Erzeugung einer sprachspezifischen Programmierschnittstelle für den Zugriff und die typkorrekte Manipulation von ASTs der verwendeten Sprache.

Die Spezifikation der abstrakten Syntax wird nach der GRAMPS-Notation durchgeführt, die Spezifikation der Präsentationsaspekte in Form der konkreten Grammatik erfolgt als Unparsing-

Schemata. Die Spezifikation von Attributwerten basiert auf einem operationalen Vorgehen, allerdings ist die Unterstützung seitens des Systems nur rudimentär ausgebildet. Als Attributwerte sind ganzzahlige Werte oder Verweise auf andere Baumknoten zugelassen.

Die Entwicklung sprachspezifischer Werkzeuge wird, wie auch die Attributierung, in der Programmiersprache BETA vorgenommen. Im folgenden wird die Komponente zur Repräsentation der ASTs und der im MPS zu verwendende Spezifikationsmechanismus näher betrachtet.

2.4.5.1 Repräsentation attributierter abstrakter Syntaxbäume

Die Komponente zur Repräsentation attributierter ASTs ist als Bibliothek in BETA realisiert. Sie arbeitet hauptspeicherbasiert, d.h. ein Syntaxbaum befindet sich während der Verarbeitung vollständig im Vordergrundspeicher. Die Persistenz der Syntaxbäume wird erreicht durch Abspeicherung im Dateisystem.

Für die interne Darstellung der Baumstruktur und die persistente Repräsentation wird ein gepacktes Format verwendet (*bytestream*). Dort liegen die eigentlichen Informationen zu den Knoten, wie Knotentyp, Verweise auf Vaterknoten und Sohnknoten, sowie die Attributwerte in serialisierter Form vor. Der Zugriff auf Knoten des Syntaxbaumes erfolgt über Schnittstellenobjekte, die Operationen für Navigation und Attributzugriff bereitstellen. Diese Schnittstellenobjekte werden durch das MPS erzeugt und verwaltet. Der Benutzer hat keine Möglichkeit durch die Verwendung gängiger objektorientierter Techniken, wie Subklassenbildung zur Spezialisierung, die Informationen in den Knoten zu erweitern.

Für den Zugriff auf ASTs bietet das MPS drei verschiedene Ebenen, die in Form von getrennten Schnittstellen realisiert sind:

Sprachunabhängiger Zugriff:
: Hier werden keine sprachspezifischen Informationen verwendet, lediglich die Baumeigenschaften finden Verwendung. Der Zugriff erfolgt über Schnittstellenobjekte, die das MPS bereitstellt (allgemeine Objekte der Klassen *Cons*, *List* usw.). Dies bedeutet insbesondere, daß eine Überprüfung der Knotentypen nicht erfolgen kann. Es können also syntaktisch inkorrekte Syntaxbäume erzeugt werden. Diese Schnittstelle wird üblicherweise von sprachunabhängigen Werkzeugen (wie dem syntaxbasierten Editor) genutzt. Sprachunabhängige Werkzeuge erhalten die Informationen zu der aktuell verwendeten Sprache in Form einer Metabeschreibung der Grammatik. Typische Operationen sind *Lesen oder Setzen des Sohns Nummer i*, *Iterierung über Listen* oder *Baumdurchläufe*.

Sprachspezifischer Zugriff:
: Es werden Schnittstellenobjekte der generierten sprachspezifischen Schnittstelle verwendet. Die Schnittstellenobjektklassen sind spezialisierte (abgeleitete) Klassen der entsprechenden MPS-Klassen, sie realisieren Typüberprüfungen und gewährleisten somit die Erzeugung syntaktisch korrekter Programme. Typische Operationen sind hier beispielsweise *Setzen oder Lesen der Bedingung einer `if`-Anweisung ausschließlich durch ein erlaubtes Objekt* oder *Zugriff auf den Anweisungsteil einer `loop`-Anweisung*.

Semantischer Zugriff:
: Diese Schnittstelle stellt Operationen zum Zugriff auf Attributwerte von Knoten bereit. Da das MPS die Attribute nur knotenweise durchnumeriert, erhält man typische Operationen wie *Setzen oder Lesen des ganzzahligen Attributs Nummer i* oder *Setzen oder Lesen des Knotenreferenz-Attributs Nummer i*.

2.4.5.2 Sprachspezifikation im Metaprogrammiersystem

Die Spezifikation der mit dem MPS zu benutzenden Sprachen basiert auf der GRAMPS-Notation. GRAMPS ist ein Akronym für *GRAmmar-based MetaProgramming Scheme* [KLLM94] und wurde in [CI84] vorgeschlagen. Die Spezifikation der Sprache erfolgt nicht streng getrennt nach abstrakter und konkreter Grammatik. Statt dessen wird die Sprache mittels einer an BNF orientierten **strukturierten kontextfreien Grammatik** spezifiziert und enthält auch terminale Symbole.

Ziele der Spezifikation sind die Generierung von Parsertabellen sowie die Generierung einer sprachspezifischen Schnittstelle durch Abbildung der Grammatikproduktionen in eine BETA-Klassenhierarchie, die bei der Realisierung sprachspezifischer Werkzeuge Verwendung findet.

Aufbau der strukturierten kontextfreien Grammatik

Die im MPS benutzte strukturierte kontextfreie Grammatik besteht im wesentlichen aus einer Auflistung von Produktionen und einem Attributdeklarationsteil. Der Attributdeklarationsteil läßt ausschließlich die Definition der Anzahl der zu einem Baumknoten zu verwaltenden Attribute zu. Dies wird auch für die persistente Speicherung verwendet.

Jede Produktion der strukturierten kontextfreien Grammatik gehört zu genau einer der folgenden vier Gruppen:

Alternative:
: Eine Alternativenproduktion definiert eine Ebene einer Hierarchie syntaktischer Typen, indem für den zu definierenden Typ die direkten Untertypen angegeben werden.

Konstruktion:
: Eine Konstruktionsproduktion definiert ein syntaktisches Konstrukt als Aggregation von syntaktischen Typen und Terminalen.

Liste:
: Eine Listenproduktion definiert Listen unbeschränkter Länge von gleichen Nichtterminalen, die durch eine beliebige Folge von Terminalen getrennt sind. Dabei wird unterschieden zwischen Listen, die mindestens ein Element enthalten müssen und Listen, die beliebig viele Elemente enthalten können.

Optionen:
: Ein Nichtterminal wird als optional wählbar definiert, indem ihm in einer Optionsproduktion das wählbare (Ziel-) Nichtterminal zugeordnet wird.

Zusätzlich werden weitere Anforderungen an die Grammatik gestellt: Jedes verwendete Nichtterminal ist definierend in genau einer Produktion zu verwenden. Jede Hierarchie syntaktischer Typen muß eine Baumstruktur besitzen, zusätzlich darf kein in einer Alternative auf der rechten Produktionsseite genannter Typ eine Liste definieren. Diese letzten zwei Einschränkungen resultieren aus der Abbildungsvorschrift der Produktionen in eine BETA-Klassenhierarchie. Falls ein Parser für die Sprache erzeugt werden soll, muß die Grammatik LALR(1)-Eigenschaften erfüllen.

Die Verwendung einer einzigen Grammatik für die Spezifikation von abstrakten Syntaxbäumen und zur Spezifikation der Sprache für die Erstellung eines Parsers führt zu gewissen Einschränkungen bei der Modellierung. So müssen beispielsweise Kettenproduktionen eingeführt werden, um LALR(1)-Eigenschaften zu erfüllen. Kettenproduktionen enthalten ausschließlich ein Nichtterminal auf der rechten Produktionsseite.

Abbildung der Produktionen in eine BETA-Klassenhierarchie

Die von den syntaktischen Typen gebildeten Typhierarchien werden in korrespondierende Klassenhierarchien abgebildet. Diese Klassenhierarchien werden an die Basisklasse für Konstruktionsknoten des MPS *Cons* angegliedert. Zusätzlich werden für alle Listenproduktionen entsprechende Klassen an die Klasse *List* des MPS angegliedert. Zur Realisierung des typsicheren Zugriffs auf die Baumknoten erhalten die generierten Klassen Methoden, die entsprechende Typüberprüfungen beinhalten.

Abbildung 7 zeigt den Ausschnitt aus einer Spezifikation zur Modellierung von Anweisungen (durch die Nichtterminale *<Stmt>*, *<Assign>*, *<ProcCall>*, *<Loop>*, *<ForLoop>* und *<WhileLoop>*), Prozedur- und Variablennamen (*<Name>*), sowie von Ausdrücken (*<Expr>* und *<ExprList>).* Die zu diesem Beispiel generierte Klassenhierarchie wird in Abbildung 8 gezeigt. Sie enthält im oberen Teil die vom MPS bereitgestellte Klassenhierarchie. Abgeleitet von der Wurzelklasse werden Unterklassen für verschiedene Arten von Lexemen, für expandierte Knoten in Form von Konstruktionsknoten oder Listen und für nicht-expandierte Knoten (Platzhalter) bereitgestellt. Die im unteren Teil der Abbildung dargestellten generierten Klassenhierarchien korrespondieren zu den in den Alternativenproduktionen spezifizierten syntaktischen Typen. So sind z.B. *Assign*, *ProcCall* und *Loop* als Unterklasse zu *Stmt* in dieser Hierarchie abgebildet. Die Blattklassen der Hierarchie werden von zu Konstruktionsproduktionen erzeugten Klassen gebildet.

Eigenschaften des Unparsers

Das MPS bietet einen schemagetriebenen, adaptiven Unparser, der auch Auslassungen unterstützt. Die Darstellung von Kommentaren kann durch eine globale Bedingung beeinflußt werden (vergleiche Abschnitt 2.4.4).

Die Unparsing-Beschreibung wird in Form eines Schemas zu jeder Knotenart der ASTs definiert. In den einzelnen Schemata werden eine Folge von Referenzen für Terminale und Nichtterminale der strukturierten kontextfreien Grammatik, sowie mögliche Positionen für Zeilenumbrüche, Einrückungen und Abstände zwischen den Worten der Ausgabe definiert. Hierbei ist es insbesondere möglich, die Adaptivität, also die Nutzung des noch in einer Zeile vorhandenen Ausgabeplatzes zu beeinflussen.

Durch zwei Arten von Blockbildung innerhalb der Schemata kann die Verwendung von Zeilenumbrüchen durch den Unparsing-Algorithmus gesteuert werden:

- Falls ein **konsistenter Block** nicht in die aktuelle Zeile paßt, so wird an allen definierten Umbruchstellen ein Umbruch durchgeführt.
- Paßt ein **inkonsistenter Block** nicht in die aktuelle Zeile, so wird nur der nötige Umbruch durchgeführt, die folgenden Elemente des Blocks werden, falls möglich, nicht durch Umbrüche getrennt.

Das Abweichen von dieser strengen Adaptivität, z.B. das Einfügen von unbedingten Zeilenumbrüchen oder Leerzeilen, ist mit dem Unparser des MPS nicht möglich.

```
...
<Stmt> ::| <Assign> | <ProcCall> | <Loop>;
<Loop> ::| <ForLoop> | <WhileLoop>;

<Assign> ::= <Name> ':=' <Expr>;
<ProcCall> ::= <Name> '(' <ExprList> ')';
<ForLoop> ::= 'for' <Name> ':=' <Expr> 'to' <Expr> 'do'
                                <Stmt>;
<WhileLoop> ::= 'while' <Expr> 'do' <Stmt>;

<ExprList> ::* <Expr> ','

<Expr> ::| <Name> | <Value> | ...;
<Name> ::= <NameAppl>;
<Value> ::= <Const>;
...
```

Abbildung 7 Ausschnitt einer Sprachspezifikation mit dem MPS

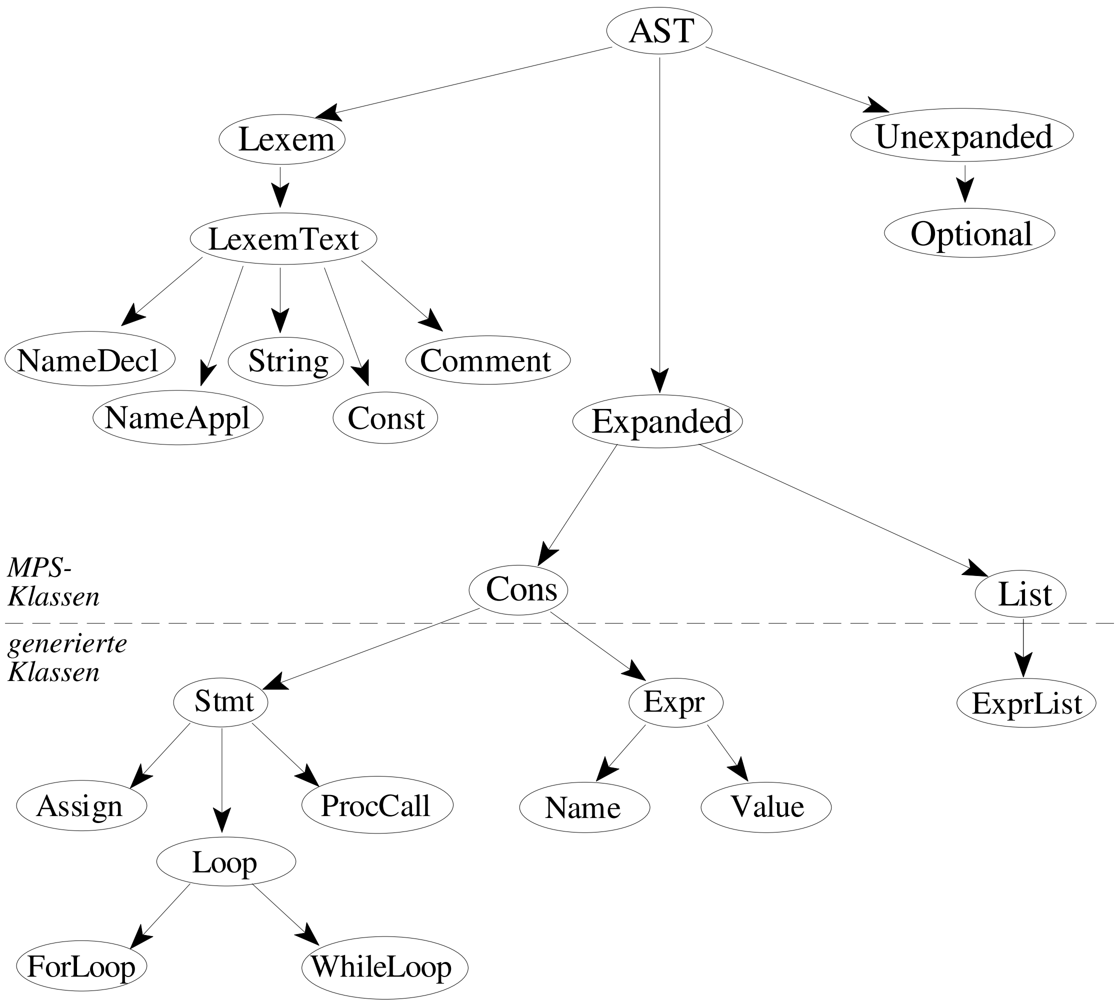


Abbildung 8 Beispiel einer generierten Klassenhierarchie

3 Anforderungsanalyse

In diesem Kapitel werden Anforderungen an die zu erstellende grammatikbasierte Prototyping-Umgebung erarbeitet. Ziel ist nicht die formale Definition der Anforderungen, vielmehr wird für Entwicklungsumgebungen allgemein und insbesondere für eine PROSET-basierte PU sinnvolle und wünschenswerte Funktionalität informell zusammengestellt.

Der erste Abschnitt dieses Kapitels beschäftigt sich in der Problembeschreibung mit einer grundlegenden Einordnung der Problemstellung und zeigt neben einem vorläufigen Systementwurf die Verwendung einer PU in Form einer Reihe von Benutzungsszenarien auf. Die folgenden Abschnitte beschäftigen sich mit den allgemeinen Anforderungen an eine grammatikbasierte PU, mit den verschiedenen Dimensionen der Integration der Umgebung und abschließend mit Anforderungen an die einzelnen Werkzeuge der Umgebung.

3.1 Problembeschreibung

Wie bereits in den Abschnitten 2.2.2 und 2.2.3 erläutert wurde, ist die Erstellung einer SEU eine sehr komplexe Aufgabe. SEUs basieren in der Regel auch auf wohldefinierten Methoden. Im Bereich des Prototyping sind zwar Methoden für die erfolgreiche Durchführung des Prototyping erarbeitet worden [BKKZ92, BP92], jedoch ist die Verwendung von PROSET nicht auf eine bestimmte Methode eingeschränkt. Aus diesem Grund ist die Erstellung einer PU in Form einer Programmierumgebung zur Unterstützung des Prototyping sinnvoll.

Auch im Rahmen der Erstellung einer Programmierumgebung zur Unterstützung des Prototyping können wünschenswerte Anforderungen aufgezeigt werden. So erfordert z.B. die schnelle Erstellung von ausführbaren Prototypen ein leistungsfähiges, auf die Eigenschaften der Prototyping-Sprache angepaßtes **Editierwerkzeug**. Hier bietet die Syntaxbasierung eine geeignete Möglichkeit zur Unterstützung.

Die Syntaxbasierung erfordert Werkzeuge, welche die Umsetzung zwischen der textuellen Darstellung und der syntaxbasierten Repräsentation vornehmen, dies sind einerseits ein **Parser** für die Konvertierung der Textdarstellung in eine syntaxbasierte Repräsentation und andererseits ein textueller **Unparser** für die umgekehrte Richtung.

Die Notwendigkeit der Prüfung der Konsistenz von Dokumenten führt im Rahmen der PU zu der Erstellung eines Werkzeuges zur **statischen Analyse** von PROSET-Programmen.

Der Wunsch nach Verwendung von externen, bereits existierenden oder zukünftigen Werkzeugen zusätzlich zu den oben beschriebenen, führt zu der Frage nach der Integration der verschiedenen Werkzeuge. Dabei muß die Integration in mindestens den drei Dimensionen Datenintegration, Präsentationsintegration und Kontrollintegration gewährleistet sein (siehe Abschnitt 3.2).

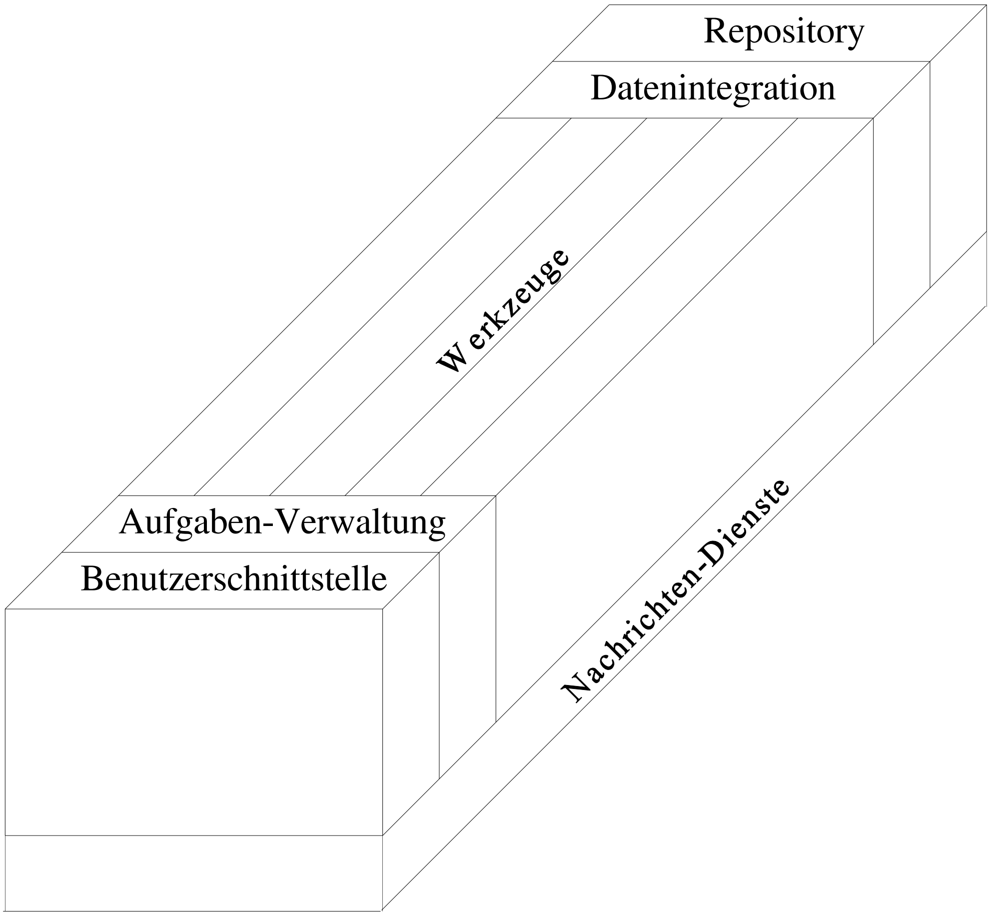


Abbildung 9 ECMA-Referenzmodell für Software-Entwicklungsumgebungen

3.1.1 Vorläufiger Systementwurf

Als Diskussionsbasis für die Anforderungsanalyse, insbesondere auch für das Verständnis der Benutzungsszenarien, wird das Referenzmodell für Software-Entwicklungsumgebungen, das durch die ECMA (*european computer manufacturer association*) vorgeschlagen wurde (vergleiche auch [Balz93]), als ein vorläufiger Systementwurf verwendet.

Das in Abbildung 9 dargestellte ECMA-Referenzmodell wurde als Vergleichsarchitektur zur Einordnung vorhandener Software-Entwicklungsumgebungen entwickelt. Es strukturiert Entwicklungsumgebungen in Teilsysteme zur Datenhaltung, Präsentation und Kommunikation und einzelne Werkzeugkomponenten, die in "Einschüben" (*slots*) zwischen Datenhaltung, Präsentationskomponente und Kommunikationskomponente eingepaßt sind.

Als vorläufiger Systementwurf wird hier eine nach dem ECMA-Referenzmodell strukturierte Umgebung mit den Werkzeugen Parser, Editor, Unparser und statische Analyse betrachtet. Die AST-Verwaltung dient der Verwaltung der abstrakten Syntaxbäume und nimmt den Platz von Datenintegration und Repository des Referenzmodells ein. Die AST-basierten Werkzeuge der Umgebung nutzen neben den Diensten der Datenhaltung auch Dienste der Benutzerschnittstelle zur Kommunikation mit dem Benutzer. Die Aufgabenverwaltung des Referenzmodells entfällt, da keine prozeßgesteuerte Umgebung betrachtet wird. Die Koppelung zusätzlicher Werkzeuge

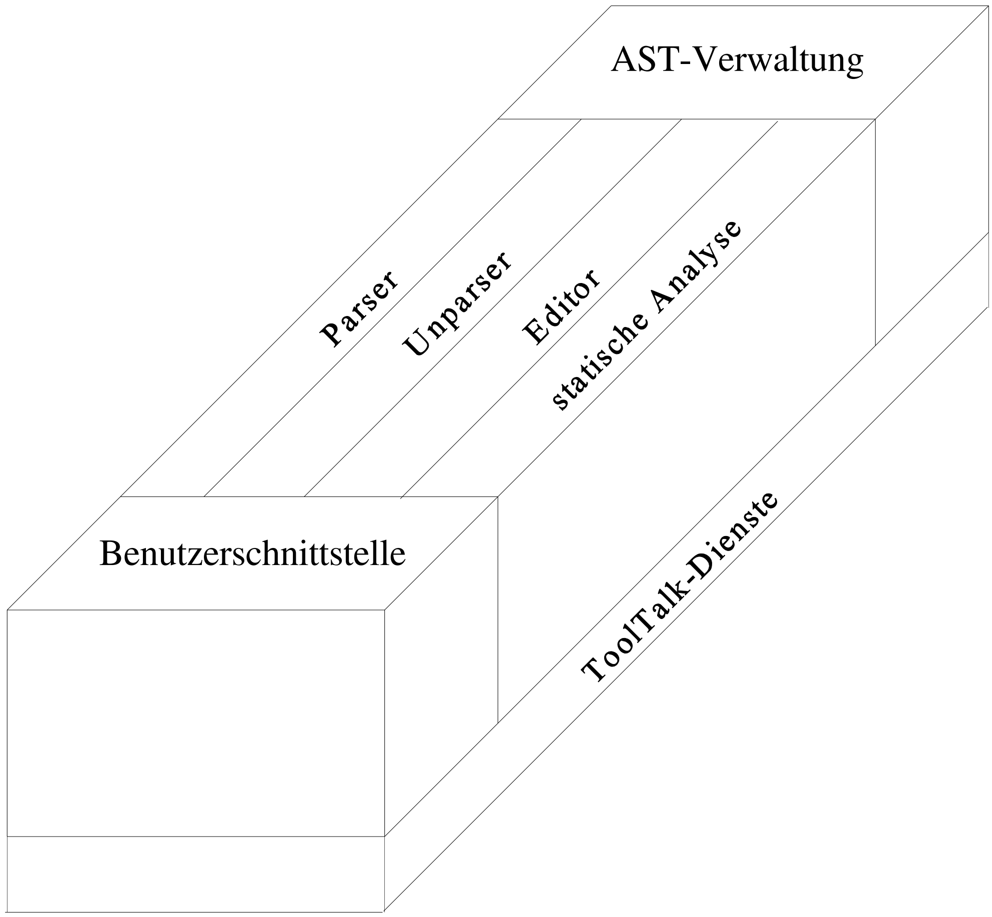


Abbildung 10 Vorläufiger Systementwurf

mit eigener Datenhaltung und Präsentationsschicht wird über die ToolTalk-Dienste zur Verfügung gestellt. Abbildung 10 zeigt den beschriebenen vorläufigen Systementwurf.

3.1.2 Benutzungsszenarien

Dieser Abschnitt beschreibt die Arbeit mit einer PU anhand von kurzen Benutzungsszenarien. Sie sollen dem Leser einen ersten Eindruck der Zusammenarbeit der beschriebenen Komponenten vermitteln, sind für das Verständnis der Arbeit nicht unbedingt notwendig, so daß sie von Lesern, die mit der Thematik vertraut sind, auch übersprungen werden können.

Öffnen eines Programms
Der Benutzer öffnet ein Programm zur Bearbeitung mit der Umgebung über Funktionen der Benutzerschnittstelle. Daraufhin wird das Programm über die Komponente zur AST-Verwaltung zur Verarbeitung bereitgestellt. Das Programm wird in einem Editorfenster angezeigt, das eine vom Unparser erzeugte, lesbare Darstellung enthält.

Syntaxbasiertes Editieren
Der Benutzer kann innerhalb des Programms navigieren und die Selektion von Inkrementen des Programms vornehmen. Die über die Benutzerschnittstelle bereitgestellte Funktionalität des Editors bezieht sich auf das selektierte Inkrement. Der Benutzer kann das aktuelle Inkrement kopieren

oder ausschneiden, im Falle von Platzhaltern können Einfügungen oder Expandierungen vorgenommen werden.

Textuelles Editieren
Der Benutzer kann das selektierte Inkrement auch textuell editieren. Die textuelle Editierung erfolgt mit herkömmlichen zeichen- oder zeilenbasierten Kommandos. Der Abschluß der textuellen Editierung wird durch den Benutzer durch ein explizites Kommando veranlaßt. Daraufhin wird der modifizierte Text durch den Parser analysiert und anschließend in den AST übernommen. Ist der modifizierte Programmteil syntaktisch fehlerhaft, so werden diese Fehler angezeigt, und dem Benutzer wird die Möglichkeit gegeben diese zu beseitigen.

Statische Analyse des Programms
Der Benutzer verwendet das Werkzeug zur statischen Analyse des Programms, um über vorhandene semantische Fehler informiert zu werden. Die Fehler werden in einem Fenster dargestellt, so daß der Benutzer die Behebung der Fehler vornehmen kann.

Verwendung von Querverweisen
Der Benutzer kann die durch die statische Analyse bereitgestellten Querverweise nutzen, um innerhalb des Baumes zu navigieren. So kann der Benutzer zu Namensanwendungen die zugehörige Deklaration erreichen und umgekehrt.

Aufruf externer Werkzeuge
Der Aufruf von externen Werkzeuge kann durch den Benutzer mittels Kommandos der Benutzerschnittstelle initiiert werden. Diese Kommandos werden durch die ToolTalk-Dienste an das betreffende externe Werkzeug übermittelt. Beispielsweise kann der Benutzer Informationen über persistente PROSET-Werte erhalten.

3.2 Integration der Werkzeuge

Die Prototyping-Umgebung soll sich einerseits durch ein einheitliches Erscheinungsbild für den Benutzer, andererseits durch eine einheitliche Repräsentation der zu verwaltenden Informationen auszeichnen. Hierbei spielt der Integrationsbegriff eine entscheidende Rolle. Nachfolgend wird deshalb eine Begriffsbestimmung für Integration nach Thomas und Nejmeh [TN92] gegeben.

Integration beschreibt die Eigenschaften der Relation zweier Werkzeuge/Dienste zueinander. Integration eines Werkzeuges oder Dienstes mit einem anderen beschreibt, daß beide gut aufeinander abgestimmt sind und als Einheit erscheinen. Integration gibt dabei den Grad des Einvernehmens zweier Werkzeuge/Dienste bezüglich gewisser Kriterien an, wie z.B. der Syntax und der Semantik der verwendeten Daten und deren gemeinsamer Nutzung oder auch der Benutzung gleicher Interaktionsmetaphern für ähnliche Bedienfunktionen. Im Hinblick auf die Integration eines Werkzeuges mit einer Umgebung (einer Gruppe aus mehreren Werkzeugen/Diensten) wird der binäre Relationsbegriff erweitert. Integration umfaßt dann alle binären Integrationsaspekte und besagt, daß das Werkzeug sich in die Umgebung einpaßt, mit allen weiteren Werkzeugen und mit den sonstigen Diensten der Umgebung bezüglich der betrachteten Kriterien in hohem Maße integriert ist. Dieses Werkzeug arbeitet als Teil des kohärenten Ganzen.

Im Vordergrund steht hierbei die Sichtweise des Benutzers: Aus seiner Sicht ist eine Umgebung integriert, wenn die konsistente Bedienung sowie angemessene Antwortzeiten gewährleistet sind. Die Sichtweise des Entwicklers ist eine andere. Er bezeichnet zwei Werkzeuge/Dienste als leicht integrierbar, wenn der Aufwand für deren Integration gering ist.

Integration wird im folgenden im Hinblick auf die Haltung der internen Repräsentation (Datenintegration), auf die Gestaltung der Mensch-Maschine-Schnittstelle (Präsentationsintegration) und die Kooperation der Werkzeuge (Kontrollintegration) untersucht.

3.2.1 Datenintegration

Die PU soll als Basis zentrale Dienste zur Repräsentation der bei der Software-Entwicklung anfallenden Dokumente enthalten. Im folgenden werden die Anforderungen innerhalb von Software-Entwicklungsumgebungen und Programmier- beziehungsweise Prototyping-Umgebung getrennt untersucht.

Aufgaben innerhalb von Software-Entwicklungsumgebungen

Innerhalb einer SEU ist die Hauptaufgabe der Datenhaltung die Verwaltung aller im Software-Entwicklungsprozeß anfallenden Dokumente in integrierter Form. Hierzu gehören neben der reinen Speicherung der Dokumente und deren Abhängigkeiten auch zusätzliche Verwaltungsinformationen wie Zugriffsrechte. Auf einem höheren semantischen Niveau für die Handhabung der Informationen werden außerdem die Verwaltung verschiedener Versionen und Konfigurationen des Informationsbestandes gefordert.

Im Hinblick auf die Unterstützung der Entwicklung komplexer Software-Systeme in Teamarbeit ist für die Datenhaltungskomponente unbedingt die Mehrbenutzerfähigkeit, die Bearbeitung von Dokumenten durch verschiedene Benutzer zur gleichen Zeit erforderlich. Für die Software-Entwicklung müssen ebenfalls erweiterte Transaktionsmechanismen, wie z.B. lange Transaktionen [BK91] durch die Datenhaltung bereitgestellt werden.

Aufgaben in Programmier- und Prototyping-Umgebungen

In Programmier- und Prototyping-Umgebungen besteht die Hauptanforderung an die Datenhaltung in der persistenten Speicherung von attributierten abstrakten Syntaxbäumen als Repräsentation von Programmdokumenten und Zugriffsoperationen auf diese Repräsentation. Es müssen sowohl Operationen für lesenden Zugriff und Navigation innerhalb der Baumstruktur, als auch für schreibenden Zugriff wie Einfügen und Löschen von Knoten und Attributen angeboten werden. Hierbei ist zu beachten, daß die zu repräsentierenden Syntaxbäume in ihrer Größe nicht beschränkt sind.

In Programmier- und Prototyping-Umgebungen ist besonderes Augenmerk auf die Effizienz des Zugriffs insbesondere bei der Editierung von Dokumenten zu legen, um lange Wartezeiten zu vermeiden. Bei der Editierung findet der überwiegende Anteil an modifizierenden Zugriffen statt, wobei die Änderungen üblicherweise lokalen Charakter haben und nicht den gesamten Syntaxbaum betreffen.

Insbesondere sind auch in Programmier- und Prototyping-Umgebungen Mehrbenutzerfähigkeit und Versionsverwaltung wünschenswert, so daß eine verteilte Entwicklung unterstützt werden kann.

Im folgenden sollen einige Kriterien für ein hohes Maß an Datenintegration aufgeführt werden [TN92]. Zentraler Gesichtspunkt ist die Konsistenz der verwalteten Informationen:

Interoperabilität: Die gleiche Sicht von Werkzeugen auf gemeinsam genutzte Daten vermeidet die Notwendigkeit aufwendiger Konvertierungen. Dies ist ein Kriterium guter Integration aus der Sicht des Entwicklers. Aber auch bei unterschiedlicher Sicht auf diese Daten kann das Maß der Integration für den Benutzer hoch erscheinen, falls ein Konvertierungsprozeß nicht explizit angestoßen werden muß und auch keine langen Wartezeiten bei der Konvertierung entstehen.

Nichtredundanz: Die Verwaltung redundanter Daten sollte vermieden werden. Redundante Daten können durch Duplizierung in verschiedenen Werkzeugen entstehen oder durch Verwaltung von Daten, die aus denen anderer Werkzeuge ableitbar sind. Der Verzicht auf die Verwendung redundanter Daten führt zu guter Integration bezüglich diesen Kriteriums, da keine Probleme der Konsistenzerhaltung auftreten.

Konsistenz: Auch bei Verzicht auf die Verwendung redundanter Daten ist es erforderlich gewisse semantische Abhängigkeiten zwischen Daten aufrecht zu erhalten. Das Integrationskriterium Konsistenz beschreibt dabei die Qualität der Unterstützung der Werkzeuge bei Änderungen von Daten, für die semantische Abhängigkeiten bestehen. Teilen die beteiligten Werkzeuge sich Änderungen gegenseitig mit, so daß diese zur Erhaltung der Konsistenz entsprechend reagieren können, so ist die Integration bezüglich der Konsistenz gut.

Im folgenden wird eine weitere Dimension der Integration, die Präsentationsintegration, betrachtet.

3.2.2 Präsentationsintegration

Ziel der Präsentationsintegration ist die Effektivitätssteigerung der Benutzerinteraktion mit der Umgebung durch einheitliche Gestaltung der Benutzeroberfläche und Verwendung gleicher Metaphern zur Interaktion, sowie die Optimierung häufig durchgeführter Bedienabläufe.

Angestrebt wird hier gemäß [TN92] eine leichte Erlernbarkeit der Bedienung, die erreicht werden kann durch:

- Beschränkung auf eine geringe Anzahl von Interaktions- und Präsentationsmechanismen
- Orientierung an den vom Benutzer selbst verwendeten Denkmodellen
- Einhaltung der vom Benutzer erwarteten Antwortzeiten
- Verfügbarkeit von für den Benutzer relevanten Informationen

Des weiteren wird in [ES89] die Modifreiheit als ein wesentliches Charakteristikum für Benutzerfreundlichkeit herausgestellt. Ein Modus ist dabei ein Zustand der Umgebung, in dem nur ein eingeschränkter Leistungsumfang für den Benutzer zur Verfügung steht, insbesondere ist dies die Einschränkung auf ein einziges Werkzeug der Umgebung (z.B. Editiermodus, Übersetzungsmodus oder Ausführungsmodus). Schließen sich durch die Arbeit in einem Modus Werkzeuge gegenseitig aus, so führt das natürlich zu Nachteilen in der Unterstützung des Benutzers. Hinzu kommt, daß diese Modi üblichweise durch explizite Kommandos ein- bzw. ausgeschaltet werden müssen. Auszunehmen sind hier Situationen, in denen bewußt auf die Unterstützung seitens der Umgebung verzichtet wird, wie z.B. freie Editierung und die nicht-syntaxbasierte Eingabe von Bezeichnern und Literalen.

Verbreitete Standard-Fenstersysteme (z.B. Motif oder OpenLook im UNIX-Bereich) bieten zwar eine gewisse Basis für die einheitliche Gestaltung der Oberfläche, jedoch erhält man erst durch die Verwendung von weitergehenden Standardisierungstechniken (*style guides*) gute Ergebnisse für die Präsentationsintegration. Eine Untersuchung bezüglich der Standardisierungstechniken und die Ausarbeitung von Oberflächenstandards für die Werkzeuge einer Prototyping-Umgebung für PROSET wurde in [Merl96] vorgenommen.

3.2.3 Kontrollintegration

Damit die Werkzeuge und Dienste einer Umgebung gemeinsam zur Erfüllung einer Aufgabe verwendet werden können, müssen diese nicht nur im Hinblick auf Daten- und Präsentationsaspekte aufeinander abgestimmt sein. Die Werkzeuge müssen kooperieren und somit miteinander kommunizieren können, damit Funktionalität des einen Werkzeuges durch ein anderes genutzt werden kann. Ziel der Kontrollintegration ist die flexible Verbindung der verschiedenen Werkzeuge zur Erfüllung der Aufgabe.

Im Rahmen der Kontrollintegration ist die Kommunikation von Werkzeugen im Hinblick auf verschiedene Aspekte zu unterscheiden:

Repräsentation: Es müssen einerseits Werkzeuge verwendet werden, die auf der gleichen Repräsentation basieren, andererseits kann die verwendete Repräsentation unterschiedlich sein.

Prozeßkontext: Neben Werkzeugen, die in einem gemeinsamen Prozeßkontext ablaufen, müssen auch Werkzeuge über Prozeßgrenzen hinweg kommunizieren können.

Sitzung: Für die Erfüllung einer Aufgabe sollten ausschließlich die Werkzeuge der eigenen Sitzung verwendet werden, die Verwendung von Werkzeugen anderer Sitzungen ist auszuschließen.

Im Hinblick auf die Kontrollintegration innerhalb der Prototyping-Umgebung ist das Vorhandensein einer Komponente zur Anbindung externer Werkzeuge hervorzuheben. Diese externen Werkzeuge basieren auf einer eigenen Repräsentation und werden in einem anderen Prozeßkontext ausgeführt. Zur deren Anbindung ist es nötig, ihnen gewisse Informationen aus dem AST mitzuteilen. Insbesondere soll die PU eine Verwendung der bereits zu PROSET entwickelten Werkzeuge, wie des PROSET-Übersetzers und der Persistenzwerkzeuge (*p-file editor* und *p-file tool*) ermöglichen. Zusätzlich sollen zukünftige Werkzeuge ebenso in die Umgebung integriert werden können.

Die Nutzung von Diensten der Prototyping-Umgebung durch andere Werkzeuge ist ebenfalls wünschenswert. Beispielsweise könnte ein zukünftiges Werkzeug zur Fehlersuche die Präsentationsdienste der PU verwenden.

3.3 Anforderungen an die Werkzeuge

Nachdem die Integrationsaspekte in den vorangegangenen Abschnitten beleuchtet worden sind, werden im folgenden die Anforderungen an die einzelnen Werkzeuge der Prototyping-Umgebung betrachtet.

3.3.1 Parser

Der Parser besitzt innerhalb der PU die Aufgabe, abstrakte Syntaxbäume zur Repräsentation von Programmtexten zu konstruieren. Die Konstruktion von Baumstrukturen aus Quelltexten ist eine im Übersetzerbau hinreichend bekannte Problemstellung. Diese Aufgabe übernehmen dort die Werkzeuge Scanner (lexikalische Analyse) und Parser (syntaktische Analyse). Im Vergleich zu der Übersetzung von Programmen, sind für eine grammatikbasierte Umgebung teilweise unterschiedliche Anforderungen seitens des Parsers zu erfüllen und gewisse Nebenbedingungen einzuhalten.

Während der Konstruktion des abstrakten Syntaxbaumes muß ebenfalls die Überprüfung des Programms auf syntaktische Korrektheit durchgeführt werden. Programme mit lexikalischen oder syntaktischen Fehlern können nicht in einen AST überführt werden. ASTs stellen nach ihrer Definition ausschließlich fehlerfreie Programme dar.

Die oben genannten Unterschiede zu traditionellen Parsern werden in den folgenden Abschnitten näher ausgeführt. Dies sind insbesondere die Forderung nach der Verarbeitung unvollständiger Programme, sowie die Notwendigkeit auch Kommentare in den AST zu übernehmen. Abschließend wird die Verwendung des Parsers innerhalb von Hybrideditoren betrachtet.

3.3.1.1 Unvollständige Programme

Programme sind unvollständig, falls sie noch Stellen enthalten, an denen noch Inkremente eingefügt werden müssen, damit sie syntaktisch korrekt werden. Dies kann zum Beispiel der Fall sein, falls noch nicht alle Teile einer Anweisung mit den erforderlichen Angaben gefüllt wurden (z.B. ist die Angabe einer Bedingung in einer `if`-Anweisung unbedingt nötig). In anderen Fällen ist die Einfügung von Inkrementen zwar möglich, aber nicht notwendig. Diese optionalen Inkremente werden zum Beispiel (je nach Modellierung der Inkrementstruktur) für den `else`-Fall von `if`-Anweisungen verwendet. Werden fehlende Inkremente durch die Angabe eines Platzhalters angezeigt, so ist die syntaktische Analyse dieser unvollständigen Programme möglich und durchaus sinnvoll. So können unvollständige Programme in der Umgebung verwendet und weiterverarbeitet werden, insbesondere können weitere Analysefunktionen zu einem frühen Zeitpunkt verwendet werden.

Aus diesem Grund soll der Parser die Übersetzung unvollständiger Programme in ASTs ermöglichen. Da im Kontext der PU der Parser die lexikalische Analyse beinhaltet, führt die Darstellung unvollständiger Programme zu einer Erweiterung der Syntax der Quelltexte, insbesondere der Definition der syntaktischen Struktur von Platzhaltern.

3.3.1.2 Übernahme von Kommentaren

In traditionellen Parsern, die im Übersetzerbau eingesetzt werden, ist die Übernahme von Kommentaren in der Regel nicht erforderlich. Auf die Ausführung eines übersetzten Programms haben sie keinen Einfluß. Innerhalb von Entwicklungsumgebungen ist eine Verwaltung von Kommentaren im AST unbedingt erforderlich. Somit müssen auch die im Programmtext vorhandenen Kommentare in den konstruierten AST übernommen werden. Dies kann durch die Bindung eines Kommentars an den Knoten des umschließenden Inkrements oder eines benachbarten Inkrements geschehen.

3.3.1.3 Verwendung zur freien Eingabe in Hybrideditoren

Die Funktionalität des Parsers wird nicht nur für die Übernahme von nicht mit der PU entwickelten Programmen benötigt, sondern kann auch zur Analyse der durch freie Eingabe innerhalb eines Hybrideditors (siehe Abschnitt 2.4.3) erstellten Programmteile dienen.

Damit der Parser die Möglichkeit bietet, die Konstruktion mit beliebigen syntaktischen Konstrukten zu beginnen, muß seine Architektur mehrfache Eintrittspunkte bieten (*multiple-entry parser*), so daß für jeden syntaktischen Typ ASTs erzeugt werden können.

3.3.2 Unparser

Der Unparser realisiert die Gewinnung einer lesbaren Darstellung für ein als AST repräsentiertes Programm. Diese lesbare Darstellung kann textuell oder graphisch oder eine Kombination von beidem sein. Im Rahmen der Prototyping-Umgebung sollte man sich zuerst auf eine textuelle Darstellung beschränken, die sinnvollerweise wieder die Darstellung in Form eines Quelltextes sein sollte.

Die konkrete textuelle Darstellung zu einem AST erhält man, indem die abstrakte Struktur durch Schlüsselwörter, Begrenzer und eine geeignete Formatierung durch Festlegung von Zeilenumbrüchen und Einrückungen ergänzt wird. Diese Formatierung sollte leicht änderbar sein, so daß für die individuelle Benutzerkonfiguration alternative Darstellungen oder Anpassungen möglich sind.

An weiteren Konzepten, wie sie in Abschnitt 2.4.4 beschrieben wurden, soll der Unparser auch die Möglichkeit der Ausblendung von Teilbäumen bieten.

3.3.3 Editor

Der syntaxbasierte Editor stellt innerhalb der PU die Funktionalität zur Manipulation von Programmen in Form von ASTs zur Verfügung. Im Gegensatz zu gängigen Editierwerkzeugen werden die Präsentationsaspekte hier ausgenommen. Die Präsentation der ASTs erfolgt über den im vorhergehenden Abschnitt beschriebenen Unparser. Der Editor umfaßt die Funktionalität sowohl für die Erstellung von Programmen durch sukzessive Ersetzung der Platzhalter durch konkrete Konstrukte der Sprache (das Anlegen von Teilbäumen), als auch die Funktionalität für die Änderung von Programmen durch Kopieren, Löschen und Einfügen von Teilbäumen.

Transformationen

Zusätzlich sollen noch sprachspezifische **Transformationen** angeboten werden. Als Transformationen seien hier Editieroperationen bezeichnet, die sich aus den genannten Operationen Anlegen von Knoten, Kopieren, Löschen und Einfügen zusammensetzen. Diese Transformationen sollen einerseits die Editierung vereinfachen. Andererseits können durch Transformationen auch Editieroperationen ermöglicht werden, die sonst nicht möglich wären (z.B. sind Tauschoperationen mit ausschneiden und einfügen nur dann möglich, wenn die verwendete Zwischenablage Platz für mehrere Einträge bietet). Transformationen für die Unterstützung des Editierens können die Semantik von Programmen durchaus ändern. Dagegen sind auch semantische Transformationen denkbar, welche die Semantik des Programms beibehalten (z.B. die Überführung unterschiedlicher Schleifenkonstruktionen ineinander).

Die von einem Editor angebotenen Transformationen sollten einerseits das Editieren unterstützen, andererseits sollten einfache semantische Transformationen bereitgestellt werden. Die Bereitstellung komplexer semantischer Transformationen (z.B. das Abrollen von Schleifen oder die Umwandlung impliziter Iterationen in explizite Schleifen) sollte nicht einem Editierwerkzeug, sondern einem dedizierten, in die Umgebung integrierten Transformationswerkzeug zugeordnet werden.

Vor- und Nachteile syntaxbasierter Editoren

Ein Hauptvorteil syntaxbasierter Editoren ist, daß durch ihre Verwendung die kontextfreie Korrektheit des Programms während der Editierung erhalten bleibt. Insbesondere werden Einfügungen von Teilbäumen nur korrekt bezüglich ihres syntaktischen Typs durchgeführt. Ein weiterer Vorteil besteht in der kontextbasierten Führung des Benutzers, das heißt der Editor sollte ausschließlich Kommandos anbieten, die im aktuellen Kontext erlaubt sind.

Aus den vorgenannten Eigenschaften leitet sich eine besondere Eignung für Anwender ab, die eine Sprache eher selten benutzen oder neu erlernen.

Das Editieren mittels syntaxbasierter Editoren birgt demgegenüber auch verschiedene Nachteile, von denen einige im folgenden aufgeführt sind [ES89, KLLM94]:

Behinderung:
: Fachleute nehmen gewisse Inkonsistenzen während der Entwicklung in Kauf. Sie fühlen sich durch die "Führung" des Editors eher behindert als unterstützt.

Umständlichkeit:
: Syntaxbasierte Eingabe kleiner Inkremente kann umständlich sein. Zum Beispiel ist die Eingabe von arithmetischen Ausdrücken nur durch die Auswahl vieler Befehle zu erreichen.

Verschachtelung:
: Änderung tief verschachtelter Strukturen kann umständlich sein. Auch hier sind komplexe Ausdrücke als Beispiel zu nennen: die Umstrukturierung eines Ausdrucks ist auch hier üblicherweise nur durch die Anwendung zahlreicher Befehle zu erreichen.

Gewöhnungsbedarf:
: Umstieg von traditioneller Editierung wird allgemein als gewöhnungsbedürftig empfunden und führt zu geringer Akzeptanz dieser Editoren.

Ein Ansatz zur Überwindung der Nachteile ist die Realisierung des Editors als Hybrideditor (siehe Abschnitt 2.4.3). Durch die Verwendung der syntaxbasierten Eingabe können die Vorteile der Benutzerführung und die Unterstützung bei der Erstellung korrekter Programme genutzt werden, sofern dies gewünscht wird. Der Benutzer kann jedoch bei Bedarf die textuelle Eingabe dort verwenden, wo sie einfachere Handhabung verspricht.

Detaillierte Untersuchungen über sinnvolle Funktionalität eines syntaxbasierten Editors für PROSET, der eine geeignete Benutzerschnittstelle bereitstellt, werden im Rahmen einer weiteren Diplomarbeit durchgeführt [Bubo96].

3.3.4 Statische Analyse

Wie in den vorhergehenden Abschnitten beschrieben wurde, stellen Editor und Parser der PU die kontextfreie Korrektheit der erstellten Programme sicher. In der Literatur existieren unterschiedliche Definitionen bezüglich der syntaktischen Korrektheit [WM92, WG94]. In dieser Arbeit soll die syntaktische Korrektheit neben der kontextfreien Korrektheit jedoch noch weitere, nicht kontextfrei beschreibbare Eigenschaften umfassen. Die Überprüfung dieser kontextsensitiven Einschränkungen ist eine der Aufgaben des Werkzeuges der statischen Analyse. Eine weitere Aufgabe des Werkzeuges ist die Bereitstellung zusätzlicher Analysefunktionen, die den Entwickler bei der Erstellung von Programmen unterstützen, dies sind insbesondere Funktionen, die dem Entwickler Informationen der Symboltabelle zugänglich machen.

3.3.4.1 Kontextsensitive Einschränkungen

Zur Sicherstellung der kontextsensitiven Korrektheit von PROSET-Programmen sind folgende Überprüfungen in den ASTs durchzuführen:

Namensanalyse:
Im Bereich der Namensanalyse sind die in Abschnitt 2.3.1.11 beschriebenen Sichtbarkeitsregeln anzuwenden und einige Beschränkungen bei der Deklaration von Namen zu überprüfen.

Typanalyse:
Eine statische Typanalyse, wie sie in stark typisierten Sprachen durchgeführt wird, ist wegen der schwachen Typisierung für PROSET-Programme nicht durchführbar. Alle Typüberprüfungen müssen zur Laufzeit durchgeführt, d.h. dynamisch analysiert werden und bleiben einem Laufzeitsystem für PROSET-Programme vorbehalten.

Allgemeine Kontextbedingungen:
Die Verwendung gewisser Anweisungen, Ausdrücke, Bezeichner und anderer Sprachmittel darf nur in vorgeschriebenen Kontexten erfolgen.

Kontext von Prozeduraufrufen:
Das Profil von Prozeduraufrufen muß anhand des AST mit dem Profil der Prozedurdefinition verglichen werden.

Sonstige Kontextbedingungen:
Hier müssen z.B. die Verwendung von Konstanten und gewisse Implementationsbeschränkungen überprüft werden.

Namensanalyse

In der Namensanalyse werden Deklarationen von Bezeichnern in Beziehung gesetzt zu Anwendungen von Bezeichnern. Dies schließt insbesondere den Aufbau einer Symboltabelle ein. Die Namensanalyse wird durch die in Abschnitt 2.3.1.10 beschriebenen Eigenschaften von PROSET beeinflußt. Im Rahmen der Namensanalyse ist die Unterstützung der verschiedenen in Abschnitt 2.3.1.11 aufgeführten Namensräume zu berücksichtigen.

Zu überprüfen sind folgende Einschränkungen:

Doppeldeklarationen innerhalb eines Bereiches:
Hier muß die Eindeutigkeit von Bezeichnern im Deklarationsteil und der Namen von Prozedur-, Behandlungsroutinen- und Moduldeklarationen überprüft werden. Hierzu gehören auch die Bezeichner innerhalb der Parameterlisten von Prozeduren und Behandlungsroutinen und die Bezeichner in Importlisten von Modulen. Zusätzlich ist die Exportliste von Modulen auf die doppelte Verwendung von Bezeichnern zu untersuchen.

Doppeldeklarationen innerhalb der Importliste von Modulinstanziierungen:
Innerhalb der Importlisten von Modulinstanziierungen werden neue Namen eingeführt. Diese Namen müssen paarweise verschieden sein.

Doppeldeklarationen von Marken innerhalb einer Schleifenanweisung:
Die Marke eines Schleifenkonstruktes darf nicht gleich derjenigen einer umschließenden Schleife sein.

Identität der Namen in den Kopf- und Fußzeilen von Programmkonstrukten:
Die in Programm-, Prozedur-, Behandlungsroutinen- und Moduldeklarationen, sowie in Schleifenanweisungen mit Marken verwendeten Namen in der Kopfzeile müssen mit den Namen innerhalb der Fußzeilen übereinstimmen.

Allgemeine Kontextbedingungen

In PROSET dürfen gewisse Anweisungen nur in bestimmten Kontexten benutzt werden. Die Verwendung der Anweisungen `quit` und `continue` ist auf Schleifen beschränkt. Die `return`-Anweisung darf ausschließlich in Prozeduren und Behandlungsroutinen verwendet werden. Die Anweisungen `resume` und `return commit` sind auf Behandlungsroutinen beschränkt. Der rekursive Aufruf von anonymen Prozeduren mittels der `self`-Anweisung darf nur in `lambda`-Konstrukten verwendet werden. Weiterhin ist die Beschränkung von Ausdrücken mit `into` auf `meet`-Anweisungen und die ausschließliche Benutzung des Ausdrucks `$` innerhalb von Ausdrücken mit `into` oder in dem Bedingungsteil von Mustern in Tupelraumabfragen (*template restriction*) zu gewährleisten.

Die Namen von Prozeduren, Modulen und Behandlungsroutinen dürfen nur in bestimmten Kontexten verwendet werden. Neben dem Vorkommen in der Fußzeile der jeweiligen Programmeinheit sind das die im folgenden beschriebenen Konstrukte.

Prozedurnamen dürfen nur in `closure`-Ausdrücken, als Modulexport oder in Prozeduraufrufen verwendet werden. Hier gilt auch die Benutzung als benutzerdefinierter binärer Operator als Prozeduraufruf. Modulnamen dürfen ausschließlich im Kontext eines `closure`-Ausdrucks oder in Modulexporten verwendet werden. Die Namen von Behandlungsroutinen dürfen dagegen nur in Ausnahmeassoziationen verwendet werden.

Die in `closure`-Ausdrücken und Ausnahmeassoziationen verwendbaren Namen sind damit vollständig festgelegt. Eine weitere Einschränkung betrifft die Art der Prozeduren, auf die das `closure`-Konstrukt angewendet werden kann: Diese dürfen ausschließlich `rd`-Parameter verwenden.

Die Exportklausel von Moduldefinitionen ist zu überprüfen. Dort dürfen ausschließlich Bezeichner von Prozeduren oder Modulen verwendet werden, die in diesem Modul definiert wurden.

Schließlich ist die Verwendung von vordefinierten Ausnahmen in Anweisungen zur Auslösung von Ausnahmen zu überprüfen. Hier muß sichergestellt werden, daß diese vordefinierten Ausnahmen mit der richtigen Anzahl von Parametern aufgerufen werden.

Kontext von Prozeduraufrufen

Im Falle eines Prozeduraufrufes kann die Anzahl der aktuellen Parameter und die Anzahl der formalen Parameter der Prozedurdefinition überprüft werden. Ebenso kann die Gültigkeit eines aktuellen Parameters für formale `wr`- oder `rw`-Parameter geprüft werden. Es ist sicherzustellen, daß der aktuelle Parameter syntaktisch einem l-Wert entspricht.

Diese Überprüfung kann allerdings ausschließlich für Aufrufe von **Prozeduren** durchgeführt werden, also nur dann, wenn der Name eindeutig einer Prozedurdefinition zuzuordnen ist. Für alle anderen syntaktisch einem Prozeduraufruf entsprechenden Konstrukte, wie Funktionsaufrufe oder assoziative Zugriffe auf Abbildungen, Tupel- oder Zeichenkettenselektionen kann statisch keine Entscheidung über die Korrektheit getroffen werden. Insbesondere ist auch sicherzustellen, daß die Parameterliste entweder leer oder eine geklammerte Liste von Ausdrücken ist. Die allgemeinen Selektionskonstrukte (Bereichsangaben mit `..`) sind in diesem Kontext verboten.

Die beschriebenen Beschränkungen gelten auch für die Parameterliste bei Modulprozeduraufrufen. Allerdings kann für Modulprozeduraufrufe die zugehörige Prozedur nicht ermittelt werden, also entfällt hier die Überprüfung der Parameter selbst.

Prozeduren können ebenfalls als benutzerdefinierter binärer Operator verwendet werden. Diese Verwendung ist allerdings nur für Prozeduren mit genau zwei `rd`-Parametern gestattet.

Sonstige Kontextbedingungen

Weitere Einschränkungen können durch statische Analyse des AST überprüft werden. Einige davon resultieren aus Implementationsbeschränkungen des jetzigen Übersetzers.

- Konstanten müssen an der Deklarationsstelle initialisiert werden.
- Die Modifizierung von Konstanten kann nur in einigen Kontexten erkannt werden (Zuweisungen, Prozeduraufrufe). Diese Überprüfungen sollen durchgeführt werden.
- In Modulen ist die Deklaration als `visible persistent` nicht zugelassen (dies betrifft nur den Deklarationsteil des Moduls selbst, nicht die Deklarationsteile der Modulprozeduren). Dies ist eine Implementationsbeschränkung.

Bereitstellung der Analyseergebnisse

Die Bereitstellung der Analyseergebnisse kann mittels zweier unterschiedlicher konzeptioneller Ansätze erfolgen:

- inkrementelle Überprüfung
- Überprüfung auf Anforderung

Die **inkrementelle Überprüfung** führt nach jeder Änderung im AST auch die Überprüfung der semantischen Einschränkungen durch, so daß zu jedem Zeitpunkt die Korrektheit des Programms gewährleistet ist. Dies erfordert jedoch entweder das Verbieten von Fehlern, so daß der Benutzer auftretende Fehler sofort beseitigen muß oder das Zulassen und Verwalten beliebig vieler Fehler. Ein weiterer Problempunkt sind die Anforderungen an die Reaktionszeit inkrementeller Systeme. Eine praktische Einsetzbarkeit der inkrementellen Überprüfung ist nur dann gewährleistet, wenn die Antwortzeiten des Systems bei der Editierung sehr kurz sind. Die Bereitstellung inkrementeller

Überprüfung ist erstmals in dem in [RT87] beschriebenen *synthesizer generator* erfolgt. Dort wird ein inkrementeller Attributauswertungsalgorithmus verwendet.

Die Überprüfung der Einschränkungen **auf Anforderung** führt zu Arbeiten in Zyklen ohne Unterstützung durch das Analysewerkzeug und führt in der Regel zu längeren Wartezeiten während der Durchführung der Überprüfung.

3.3.4.2 Analysefunktionen

Zusätzlich zu der Sicherstellung der kontextsensitiven Korrektheit ist die Verfügbarkeit weiterer Analysefunktionen wünschenswert. Insbesondere sind dies auf der Symboltabelle von Programmen basierende Funktionen, die zum Teil in Form von Querverweisen bereitgestellt werden können:

- Ausgabe der Symboltabelle für einen Bereich: Hiermit kann der Anwender beispielsweise Schreibfehler auffinden (implizit deklarierte Bezeichner)
- Suche der Deklarationsstelle zu einem Bezeichner (*use-definition-chain*)
- Suche aller Anwendungen eines Bezeichners (*definition-use-chain*)

An weiteren Analysefunktionen wäre die Überprüfung gewisser Qualitätskriterien denkbar, beispielsweise das Auffinden von nicht erreichbarem Programmcode oder das Erkennen von Deklarationen, die nicht verwendet werden. Wichtiges Kriterium ist auch die Sicherstellung, daß Variablen vor der Verwendung initialisiert werden. Diese Qualitätskriterien können teilweise nur durch Analyse des Kontrollflusses sichergestellt werden.

3.3.5 Weitere Werkzeuge

Zur Vervollständigung der PU ist das Vorhandensein weiterer Werkzeuge unumgänglich. Es handelt sich dabei einerseits um grammatikbasierte Werkzeuge (Übersetzer oder Interpretierer, Testwerkzeug, Werkzeug zur Fehlersuche, Transformationswerkzeug), andererseits um nicht AST-basierte Werkzeuge, wie beispielsweise zur Fehlersuche in parallelen Programmen oder der Persistenzwerkzeuge.

Grammatikbasierte Werkzeuge

Grammatikbasierte Werkzeuge profitieren von der gemeinsamen Nutzung abstrakter Syntaxbäume. Dies soll hier nur kurz andiskutiert werden. Beispielsweise können Übersetzer oder Interpretierer werkzeugspezifische Attribute mit Zwischencode erzeugen. Das Werkzeug zur Fehlersuche sollte ebenfalls die Präsentationsdienste des Unparsers benutzen. Ein Transformationswerkzeug (z.B. die Übersetzung komplexer Mengenausdrücke) kann direkt den AST manipulieren und in die Umgebung integriert werden. Für diese Werkzeuge ist eine Integration derart wünschenswert, daß möglichst viele Ergebnisse von Zwischenberechnungen wiederverwendet werden können. So kann die Benutzung der verschiedenen Werkzeuge erfolgen, ohne daß lange Wartezeiten während der Neuberechnung auftreten.

Externe Werkzeuge

Nicht-grammatikbasierte Werkzeuge werden über Kommunikationsmechanismen im Rahmen der Kontrollintegration (siehe Abschnitt 3.2.3) angebunden. Da diese Werkzeuge auf anderen Dokumenttypen basieren und auch eigenständig verwendet werden können, ist eine Nutzung ihrer Dienste durch Kopplung über einen Dienst zum Nachrichtenaustausch wünschenswert.

4 Entwurf einer Prototyping-Umgebung für PROSET

Im diesem Kapitel wird der Entwurf einer Entwicklungsumgebung zur Unterstützung des Prototyping mit PROSET basierend auf dem BETA-System beschrieben.

Das BETA-System stellt eine integrierte Umgebung zur Entwicklung von BETA-Programmen zur Verfügung. Diese Umgebung umfaßt einen Parser, einen Unparser, einen Editor und Werkzeuge zur Übersetzung von BETA-Programmen und zur Fehlersuche in BETA-Programmen. Die Umgebung ist grammatikbasiert, die Programme werden als ASTs verwaltet. Das BETA-System stellt dabei ein Rahmenwerk (*application framework*) zur Verfügung, aus dem in dieser Arbeit durch Erweiterung eine Prototyping-Umgebung für PROSET entwickelt wird. Während der Parser und der Unparser im wesentlichen nur durch die Spezifikation der zugrunde liegenden Sprache angepaßt werden können, kann die Umgebung sowohl im Bereich der Benutzerschnittstelle durch Hinzufügen von Dialogen und Menüs, als auch der Einbindung zusätzlicher Werkzeuge, z.B. eines Werkzeugs zur statischen Analyse, auf die Anforderungen anderer Sprachen angepaßt werden.

4.1 Grobentwurf

Dieser Abschnitt beschreibt die Software-Architektur der Prototyping-Umgebung in Form einzelner Teilsysteme und die Abhängigkeiten zwischen diesen Teilen. Die Teilsysteme sind in drei Schichten angeordnet. Abbildung 11 zeigt den Grobentwurf der PU. Die Pfeile zwischen den Teilsystemen beschreiben die Beziehungen der Teilsysteme untereinander und bezeichnen eine “benutzt”-Beziehung.

Die Basisschicht umfaßt die folgenden Basissysteme. Neben der Benutzerschnittstelle, dem Nachrichtenvermittlungssystem und dem Betriebssystem werden durch das Teilsystem PROSET-Repräsentation die Verwaltung abstrakter Syntaxbäume für PROSET, sowie werkzeugspezifische Metainformationen für Editor, Parser und Unparser zur Verfügung gestellt.

Aufbauend auf der Basisschicht sind die Werkzeuge in einer weiteren Schicht angeordnet. Die Werkzeugschicht besteht aus den Werkzeugen Parser, Unparser, syntaxbasierter Editor, statische Analyse, sowie dem Teilsystem zur Anbindung externer Werkzeuge (ToolTalk-Subsystem). Das Teilsystem zur Bereitstellung von Transformationen ist logisch dem Editor zuzuordnen, jedoch muß im Falle des BETA-Rahmenwerks dieses Teilsystem getrennt von dem Editor an das Rahmenwerk angebunden werden.

Die gemeinsame Kontrolle bildet die oberste Schicht der Software-Architektur. Neben der Kontrolle des Rahmenwerks ist ein weiteres Teilsystem für die Anbindung der PROSET-spezifischen Teilsysteme statische Analyse, ToolTalk-Subsystem und der Transformationen notwendig.

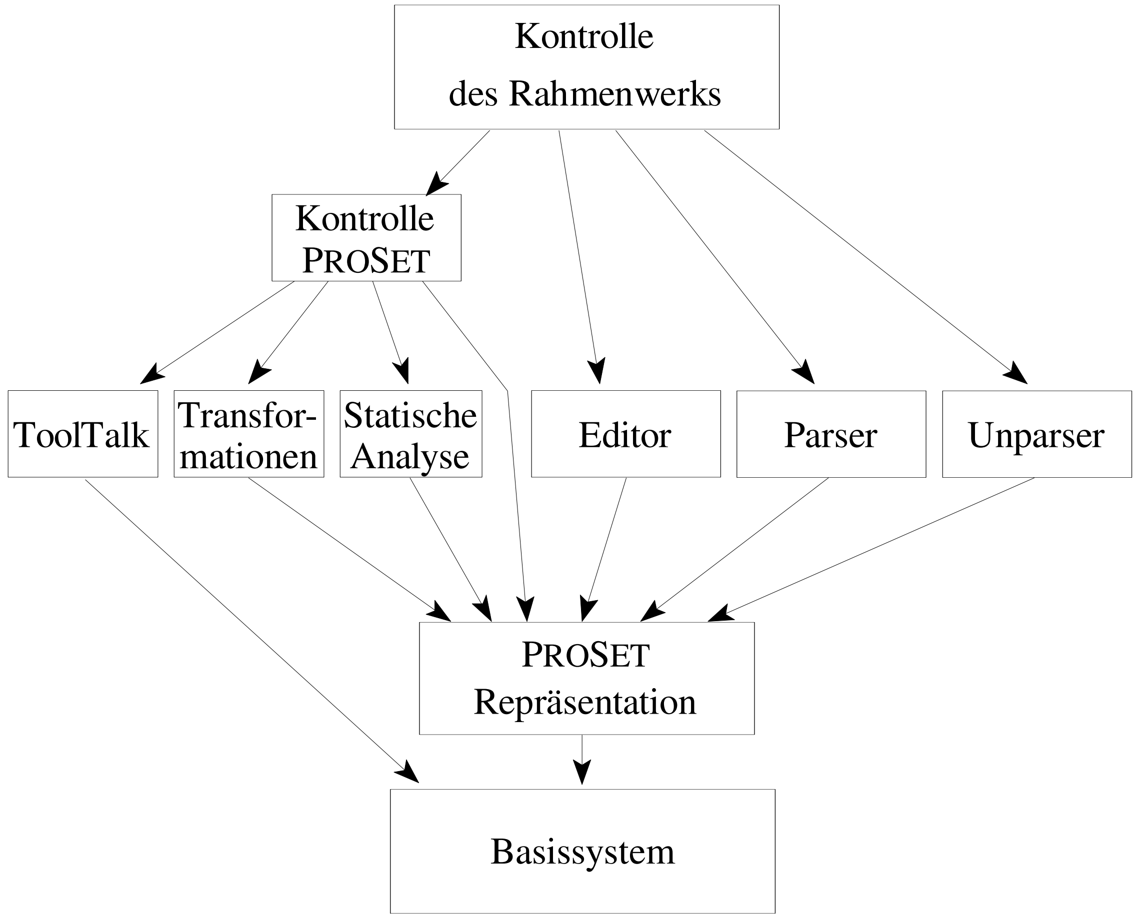


Abbildung 11 Grobentwurf der Prototyping-Umgebung

4.2 Komponenten der Prototyping-Umgebung

In den folgenden Abschnitten werden die einzelnen Teilsysteme der PU genauer betrachtet. Bei dem Entwurf jedes dieser Teilsysteme spielen die Randbedingungen des BETA-Metaprogrammiersystems und des BETA-Rahmenwerks eine wichtige Rolle.

4.2.1 Verwaltung der abstrakten Syntaxbäume

Der Entwurf der Komponente zur AST-Verwaltung kann auf Basis verschiedener Konzepte erfolgen:

- hauptspeicherbasierte Repräsentation mit persistenter Speicherung im Dateisystem des Betriebssystems
- Relationales Datenbanksystem
- Nicht-Standard-Datenbanksystem

Die **hauptspeicherbasierte Datenhaltung** und **persistente Speicherung** im Dateisystem des Betriebssystems birgt einerseits Probleme bei der Handhabung großer ASTs: Das Einlesen der persistenten Repräsentation kann zu langen Wartezeiten führen. Diese Wartezeit kann zwar durch geeignete Strategien (z.B. verzögertes Einlesen, *lazy fetch*) verkürzt werden, erfordert allerdings zusätzlichen Realisierungsaufwand. Andererseits ist auch die Bereitstellung der Mehrbenutzerfähigkeit problematisch. Dieser Ansatz wird oft von objektorientierten Systemen mit integriertem Persistenzkonzept genutzt.

Die Verwendung **relationaler Datenbanksysteme** genügt aus folgendem Grund nicht der geforderten Performanz: ASTs (aber auch sonstige Dokumente des Software-Entwicklungsprozesses) sind durch ihre stark vernetzte Struktur nur schwer auf relationale Datenbanken abzubilden. Ein Knoten wird hier auf viele Tabellen verteilt, die "Nähe" zusammengehöriger Informationen geht hierbei verloren. Bei den Zugriffen sind dann üblicherweise viele Tabellen und Verknüpfungen beteiligt, so daß gute Zugriffsgeschwindigkeiten nicht erreicht werden können. Zusätzlich zu der schlechten Eignung des Datenmodells sind auch die Transaktionskonzepte, Zugriffsschutz-, Verteilungs- und Administrationskonzepte insbesondere für SEUs ungeeignet [Kelt93].

Der komplexe und stark vernetzte Aufbau der ASTs legt die Verwendung von **Nicht-Standard-Datenbanksystemen** (NSDBS) nahe, da diese die genannten Anforderungen erfüllen können. Allerdings werden von Seiten der SEUs an NSDBS bei feingranularer persistenter Verwaltung sehr hohe Zugriffsgeschwindigkeiten gefordert, die von heutigen Systemen oftmals noch nicht erreicht werden. Hier bietet sich an, Teile der Repräsentation im Hauptspeicher bereitzuhalten oder einen grobgranulareren Ansatz zu verfolgen. Denkbar ist hier der dokumentorientierte Zugriff, es werden jeweils vollständige Dokumente gelesen und geschrieben. Die Werkzeuge arbeiten dann jeweils auf einer hauptspeicherbasierten Repräsentation dieses Dokuments. Diese Vorgehensweise übernimmt allerdings die Nachteile dateibasierter Werkzeuge, wie beispielsweise grobe Einheiten des Sperrens und der Versionierung oder Einschränkungen bei Interaktionen zwischen gleichzeitig arbeitenden Werkzeugen.

Im Rahmen dieser Arbeit wird als Verwaltung der ASTs die AST-Verwaltung des MPS verwendet. Diese AST-Verwaltung arbeitet hauptspeicherbasiert und kann die Anforderungen an Mehrbenutzerbetrieb nicht erfüllen. Die Verwendung der von BETA bereitgestellten objektorientierten Datenbank bietet sich hier zwar an, die feingranulare persistente Verwaltung der ASTs ist in Zusammenhang mit dem Metaprogrammiersystem aufgrund der verwendeten gepackten Repräsentation der ASTs, in Ermangelung einer geeigneten Schnittstelle des MPS praktisch schwer realisierbar. Hier müßten große Teile der AST-Verwaltung des MPS ersetzt werden, was in enger Zusammenarbeit und Abstimmung mit Mjølner erfolgen müßte.

Ein grobgranularer dokumentorientierter Zugriff mit Sperren ganzer Programme und der hauptspeicherbasierten Bereitstellung der Programme ist mit dem MPS durchaus denkbar. Durch die zusätzliche Einbindung externer Werkzeuge, z.B. eines Versionsverwaltungssystems, könnten so zusätzliche Anforderungen erfüllt werden. Da die Effekte in Bezug auf die Anforderungen aber als gering einzuschätzen sind, z.B. im Bereich der Rücknahme von Änderungen, wird die grobgranulare Verwaltung im Rahmen dieser Arbeit nicht weiter verfolgt.

Zur Verwendung der AST-Verwaltung des MPS ist die Spezifikation der Struktur abstrakter Syntaxbäume für PROSET in Form einer strukturierten kontextfreien Grammatik durchzuführen.

Für diese formale Spezifikation wird durch das MPS eine Klassenhierarchie generiert, die für den Zugriff auf ASTs für PROSET verwendet wird. Diese formale Spezifikation müssen auch zukünftige grammatikbasierte Werkzeuge für PROSET verwenden. Die generierte Klassenhierarchie stellt für Werkzeuge, die auf der Basis von BETA entwickelt werden, eine geeignete Schnittstelle für den Zugriff auf ASTs für PROSET dar. Zukünftige Werkzeuge müssen die benötigten werkzeugspezifischen Attribute in dieser formalen Spezifikation zusätzlich zu den bereits definierten anmelden.

Für in anderen Sprachen zu entwickelnde Werkzeuge ist der Zugriff auf ASTs über die bereitgestellte Schnittstelle problematisch.

4.2.2 Benutzerschnittstelle und Ablaufsteuerung

Die Benutzerschnittstelle wird durch die graphische Bibliothek *guiEnv* innerhalb des BETA-Systems bereitgestellt. Die Beschränkung auf die dort zur Verfügung gestellten Kontrollelemente führt zur geforderten Präsentationsintegration der Umgebung, da das Rahmenwerk den Motif *style guide* umsetzt und den Benutzer nicht auf Modi einschränkt, in denen nur die Funktionalität weniger Werkzeuge verfügbar ist. Die Forderung nach leichter Erlernbarkeit wird lediglich durch das vom Rahmenwerk zur Verfügung gestellte Tutorium und Hilfesystem erfüllt.

Die Ablaufsteuerung wird durch das BETA-Rahmenwerk vorgegeben, kann aber für zusätzlich erstellte Werkzeuge, wie beispielsweise die statische Analyse, erweitert werden.

4.2.3 Parser

Der Parser der PU basiert im wesentlichen auf dem generierten Parser des MPS und dessen Einbindung in das BETA-Rahmenwerk.

Für die Erstellung eines Parsers muß die für die Spezifikation der abstrakten Syntaxbäume erstellte strukturierte kontextfreie Grammatik die in Abschnitt 2.4.5 aufgeführten Einschränkungen erfüllen. So muß die Grammatik LALR(1)-Eigenschaften erfüllen und in eine Klassenhierarchie der Implementationssprache BETA abzubilden sein. Die zur Erfüllung der Einschränkungen durchzuführenden Schritte beeinflussen wiederum die Strukturierung der spezifizierten ASTs, z.B. durch die Einfügung zusätzlicher Knoten durch Kettenproduktionen.

Insbesondere wird für die Spezifikation arithmetischer Ausdrücke eine hierarchische Teilgrammatik erstellt. Für jede Prioritätsstufe von Operatoren wird eine getrennte Produktion verwendet. Da die Sprachbeschreibung für das BETA-MPS in einer einzigen für Parser und Editor gültigen Grammatik erfolgen muß, hat diese Repräsentation wiederum Einfluß auf die Editierung von Ausdrücken: Zur Eingabe einer Operation hoher Priorität muß für jede niedrigere Prioritätsstufe eine Auswahl erfolgen.

4.2.3.1 Format der Eingabedateien

Der Parser akzeptiert ProSet-Programme, die im wesentlichen konform zur Sprachbeschreibung [DFG+92a] sind. Allerdings gelten für das MPS einerseits im folgenden beschriebene Einschränkungen und andererseits einige zusätzliche Bedingungen.

Einschränkungen zur ProSet-Sprachbeschreibung

Die zu verarbeitenden Programme dürfen keine Makros enthalten. Das gesamte Programm muß in einer Datei enthalten sein, eine Aufteilung auf mehrere Dateien unter Verwendung des Inkludierungsmechanismus wird nicht unterstützt.

Die folgenden Punkte beschreiben Einschränkungen, die sich aus dem recht einfach gehaltenen Teilsystem zur lexikalischen Analyse innerhalb des MPS ergeben:

- ProSet gestattet die Definition von Kommentaren auf zwei verschiedene Arten: Einerseits kann kommentierender Text durch Einschließen innerhalb von `(*` und `*)` definiert werden, andererseits durch Einleitung mit `--`, das Zeilenende bildet dann das Ende des Kommentars. Das MPS unterstützt nur eine Art der Kommentarkennzeichnung. Hier wird die Verwendung der ersten Möglichkeit vorgezogen, mit dieser Notation können sowohl einzeilige als auch mehrzeilige Kommentare verwendet werden. Die Verwendung von einzeiligen Kommentaren mit `--` ist kein adäquater Mechanismus, da Zeilenorientierung sehr stark auf einem bestimmten Layout des Textes basiert und in grammatikbasierten Umgebungen durch die Verwendung von ASTs kein festes Layout für eine textuelle Darstellung garantiert werden kann.
- Das MPS unterstützt nur ein allgemeines Zahlenformat. Dieses Zahlenformat dient der Repräsentation sowohl von reellen als auch von ganzen Zahlen. Dies führt für ProSet wegen der schwachen Typisierung nicht zu Problemen.
- Da die Spezifikation von Zeichenketten innerhalb des MPS nicht mit der Spezifikation für ProSet übereinstimmt, können bestimmte Zeichenketten aus ProSet-Programmen nicht übernommen werden. Dies betrifft insbesondere Zeichenketten, die selbst das Begrenzungszeichen für Zeichenketten oder betriebssystemabhängige Kodierungen enthalten (z.B. das Zeichen `).
- Bereichsbezeichnungen mit `..` (bei Ausschnittsbildung von Tupeln und Zeichenketten) werden vom MPS-Parser nicht korrekt verarbeitet, falls die lexikalischen Einheiten nicht durch Zwischenräume getrennt sind.

Obwohl durch die rudimentäre lexikalische Analyse einige Probleme bei der Verwendung der PU zu erwarten sind, wurde nicht auf die Verwendung der bereitgestellten lexikalischen Analyse verzichtet. Die Möglichkeit ausschließlich die syntaktische Analyse zu verwenden war nicht wünschenswert, da dies zu einem starken Anstieg von Knotenarten in der AST-Verwaltung und zu einem starken Anstieg von Knoten innerhalb der ASTs führt und auch die kontextabhängig zu steuernde Verarbeitung von Leerzeichen und Zwischenräumen problematisch ist.

Zusätzliche Bedingungen für die Verwendung mit dem Parser

Für die automatische Identifizierung der Sprache, die der Eingabedatei zugrunde liegt, sowie der gespeicherten syntaktischen Struktur enthält eine MPS-konforme Quelldatei eine Kopfzeile mit dem Namen des Programms, der Bezeichnung der gespeicherten syntaktischen Struktur (Typ des Wurzelknotens) und des Namens der zugrunde liegenden Grammatik. Diese Informationen sind eingeschlossen in `--`, die Namen werden durch `:` voneinander abgegrenzt.

Platzhalter innerhalb von unvollständigen Programmen werden durch Angabe ihres syntaktischen Typs notiert. Der Name des syntaktischen Typs wird durch vorgegebene Markierungszeichen ein- und ausgeleitet (dies sind die doppelten spitzen Klammern << und >>).

Abbildung 12 zeigt ein Beispiel für eine MPS-konforme Quelltextdatei. Sie enthält eine Kopfzeile, die das Programm als Programmdefinition (syntaktischer Typ *ProgDef*) der Sprache ProSet (Grammatikname *proset*) ausweist. Der Name des Fragments für dieses Programm lautet *Test*. Des weiteren wird in dieser Datei ein Platzhalter für eine Anweisung (syntaktischer Typ *Stmt*) verwendet.

```
-- Test: ProgDef: proset --

program Test; (* MPS-konforme Quelldatei *)
begin
   <<Stmt>>;
end Test;
```

Abbildung 12 MPS-konforme PROSET-Quelldatei

4.2.3.2 Ausgabe des Parsers

Die Ausgabe des MPS-Parsers ist eine hauptspeicherbasierte Darstellung des Programms als AST. Diese Darstellung kann über die AST-Verwaltung persistent gespeichert werden.

4.2.3.3 Arbeitsweise und Benutzerinteraktion

Der Parser wird innerhalb der PU automatisch aufgerufen, wenn kein AST für ein Programm vorliegt oder die textuelle Repräsentation geändert wurde. Ist das zu verarbeitende Programm oder Programmfragment kontextfrei korrekt, so wird ein AST erzeugt, der wiederum in der Umgebung weiterverarbeitet werden kann.

Falls bei der Prüfung Syntaxfehler festgestellt werden, müssen diese ebenfalls mit Hilfe des Editors beseitigt werden. Zur Beseitigung von Fehlern wird die gesamte Eingabe zur textuellen Editierung angeboten, die Fehlerposition wird in einem gesonderten Fenster angezeigt. Der Benutzer hat hier die Möglichkeit, den Fehler zu beseitigen und anschließend den Text erneut durch den Parser analysieren zu lassen.

4.2.3.4 Importierung von PROSET-Dateien in die PU

Zur Importierung von vorhandenen (nicht MPS-konformen) Quelldateien wird ein Importfilter erstellt. Dieser Filter realisiert die Übersetzung der vorhandenen Quelltexte in das MPS-konforme Format und soll auch die oben beschriebenen Einschränkungen (soweit möglich) überwinden helfen. Da dieser Importfilter auf der textuellen Repräsentation von PROSET-Programmen arbeitet, sind die Möglichkeiten in diesem Filter auf die übliche Such- und Ersetzfunktionalität beschränkt.

Die folgenden Punkte beschreiben die Aufgaben des Importfilters:

- Inkludierungsprozessor und Makroprozessor werden optional aufgerufen, falls Inkludierungsdirektiven oder Makros enthalten sind.
- Die Quelldatei wird um eine MPS-spezifische Kopfzeile ergänzt. Als syntaktischer Typ für das MPS kommt ausschließlich *ProgDef* in Betracht. Der Name wird aus dem Dateinamen gewonnen.
- Die mit -- eingeleiteten Kommentare werden in das durch den Parser unterstützte Format umgewandelt.

Die Ausführung des Importfilters erfolgt durch expliziten Aufruf des Benutzers. Neben der Verwendung von der Kommandozeile aus, kann der Aufruf des Filters auch direkt aus der PU heraus über einen Menüeintrag erfolgen.

4.2.4 Unparser

Als Unparser der Prototyping-Umgebung wird der Unparser des MPS verwendet. Wie bereits in Abschnitt 2.4.5.2 beschrieben, arbeitet der schemagetriebene MPS-Unparser adaptiv, unterstützt Ausblendungen (*elisions*), sowie Annotationen zur Repräsentation von Kommentaren. Die Ausgabe von Attributwerten und Mehrfachsichten werden nicht unterstützt.

Als Beschreibung des Unparsing für PROSET-Sprachkonstrukte ist die Erstellung von Unparsing-Schemata für die Konstrukte der PROSET-Grammatik vorzunehmen. Die Unparsing-Schemata werden so erstellt, daß die Strukturierung von Programmen deutlich wird. Die Blockstrukturierung in Prozeduren, Module und Behandlungsroutinen soll offensichtlich werden, die feineren Strukturen - wie Deklarationen und Anweisungen - sollen möglichst zeilenweise ausgegeben werden. Feingranulare Strukturen - wie arithmetische Ausdrücke, Tupel und Mengen, Iteratoren usw. - werden dichter gepackt.

Einschränkungen bei der Realisierung der Strukturierung bildet die dem Unparser zugrunde liegende strikte Adaptivität. Das recht starre Konzept der Blockung (Abschnitt 2.4.5.2) führt dazu, daß beispielsweise keine Leerzeilen als Strukturierungsmittel oder unbedingte Zeilenumbrüche definiert werden können.

4.2.5 Editor

Als Basis für den Editor der PU wird der syntaxbasierte Editor des BETA-Rahmenwerks verwendet. Der Editor bietet Kontextmenüs mit Auswahlmöglichkeiten für zu expandierende Sprachkonstrukte. Die Beschreibung der Kontextmenüs für Expandierungen erfolgt bereits in der strukturierten kontextfreien Grammatik. Sie korrespondieren zu den dort spezifizierten syntaktischen Typen. Hierarchien von syntaktischen Typen werden im syntaxbasierten Editor als Hierarchie von Kontextmenüs verwendet. Hier konkurrieren die Anforderungen des Editors (die übersichtliche Gruppierung von Auswahlmöglichkeiten in hierarchischen Kontextmenüs) mit Anforderungen der Attributierung und des Parsers. Für die Attributierung benötigte Kettenproduktionen werden dann auch im Editor sichtbar. Der Spielraum ist durch die Zusammenfassung der Spezifikation für Parser, Editor und AST recht gering.

Für den Editor werden zusammengehörige Sprachkonstrukte, soweit dies möglich ist, auch in Kontextmenüs zusammengefaßt. Dies betrifft die folgenden Bereiche innerhalb von PROSET:

Strukturierung von Anweisungen:
PROSET enthält eine große Zahl verschiedener Anweisungen. Die Präsentation aller Auswahlmöglichkeiten auf einer Stufe behindert die Übersicht, so daß eine mehrstufige Auswahl spezifiziert wird. Hier werden logisch zusammengehörige Anweisungen - wie Schleifenkonstrukte, Operationen auf Tupelräumen usw. - jeweils in Untermenüs zusammengefaßt.

Strukturierung von Ausdrücken:
Die Strukturierung der Kontextmenüs für binäre Operationen ist vorgegeben durch die Präzedenzen der Operatoren. Um Eindeutigkeit für den Parser zu erreichen, müssen die Vorrangstufen in eine hierarchische Teilgrammatik abgebildet werden. Hierbei ist kein Gestaltungsspielraum für Strukturierung vorhanden. Auf der Stufe der einfachen Operanden werden Mengenformungsausdrücke in einem Untermenü zusammengefaßt. Gleiches gilt für Tupelformungsausdrücke.

Einfache Transformationen

Die Verwendung von Transformationen (siehe Abschnitt 3.3.3) wird nur exemplarisch erläutert und realisiert. Das MPS bietet für Transformationen keine Werkzeugunterstützung, wie diese beispielsweise für den *synthesizer generator* [RT87] angeboten wird. Die gewünschten Transformationen müssen durch Analyse des AST und Erstellen von Teilbäumen zur Ersetzung operational spezifiziert werden. Die jeweils möglichen Transformationen sollen analog zu der Auswahl bei der Expandierung innerhalb eines Kontextmenüs erscheinen.

Folgende Transformationen auf der Basis von Anweisungen werden angeboten. Der Aufruf einer Transformation erfolgt durch Anwahl eines entsprechenden Eintrages innerhalb des Kontextmenüs für das zu ersetzende Inkrement:

- Ersetzung einer `while`-Anweisung durch eine `repeat`-Anweisung und umgekehrt. Diese Transformation ist als Beispiel für die Unterstützung der Editierung (die Ersetzung eines Konstruktes durch ein strukturell verwandtes) zu betrachten, wobei die Sohnknoten beibehalten bleiben. Da es sich also nicht um ein Beispiel für eine semantische Transformation handelt, wird keine Änderung der Schleifenbedingung und des -rumpfes durchgeführt.
- Transformation von Schleifen mit Sprungmarke in Schleifen ohne Sprungmarke und umgekehrt (für `loop`, `while`, `repeat`, `for` und `whilefound`). Diese Transformation dient als Beispiel für die Ersetzung durch ein Konstrukt, dessen Struktur unterschiedlich ist. Insbesondere entfallen im Zielkonstrukt Sohnknoten, oder es werden zusätzliche Knoten erzeugt.
- Der Tausch von Anweisungslisten in `if`-Anweisungen: Die Anweisungslisten des `then`-Teils, der `elseif`-Teile und des `else`-Teils bilden eine Folge. Innerhalb dieser Folge soll es möglich sein, ein Element mit seinem Nachfolger zu tauschen. Um einen geschlossenen Zyklus zu erhalten, soll das letzte Element der Folge mit dem ersten Element getauscht werden können. Auch diese Transformation ist als Beispiel für den Tausch von Teilbäumen gedacht, wobei hier komplexere Analyseschritte innerhalb des AST durchgeführt werden müssen.
- Transformation von `case`-Anweisungen in `if`-Anweisungen und umgekehrt (analog für Ausdrücke):
  Diese Transformationen dienen als Beispiel für semantische Transformationen. Die Durchführung dieser Transformation darf die Semantik des Programms nicht verändern. Für die Transformation einer `case`-Anweisung in eine `if`-Anweisung müssen insbesondere die Bedingungen aus der `case`-Anweisung und den `when`-Teilen zu Bedingungen für `if` oder `elseif` kombiniert werden (durch Gleichsetzung). Bei Transformation einer `if`-Anweisung in eine `case`-Anweisung werden die Bedingungen aus `if` und `elseif` in die `when`-Teile übernommen. Die Grundbedingung der erstellten `case`-Anweisung lautet dann `true`.

Transformationen für den Tausch von Teilbäumen oder für Änderungen, bei denen mehrere Teilbäume beibehalten werden sollen, sind grundsätzlich erforderlich, da der MPS-Editor eine Zwischenablage mit nur einem Eintrag bietet und so mittels Ausschneiden und Einfügen kein Tausch möglich ist. Eine ausführlichere Betrachtung von Transformationen zur Unterstützung des Editierens wird in [Bubo96] vorgenommen.

4.2.6 Statische Analyse

Die statischen Analyse basiert einerseits auf einer korrekten Attributierung und andererseits auf der Überprüfung von Kontextbedingungen und Einschränkungen. Für die statische Analyse wird die Entwurfsentscheidung getroffen, daß die Überprüfung nur auf Anforderung des Benutzers durchgeführt wird. Die Überprüfung erfolgt nicht inkrementell, allerdings soll die Anzahl der Neuberechnungen durch die Wiederverwendung von Teilergebnissen möglichst gering gehalten werden. Die Entwicklung eines inkrementellen Werkzeuges erscheint in diesem Rahmen als zu komplex und unnötig, da einerseits durch das MPS keine Unterstützung der inkrementellen Berechnung geleistet wird und andererseits die Wiederverwendung von Teilergebnissen einen vertretbaren Kompromiß beschreibt.

4.2.6.1 Aufbau der Attributierung

In den folgenden Abschnitten werden die einzelnen Aspekte der Attributierung vorgestellt. Ein Ziel der Attributierung ist die geeignete Verkettung des durch die verschachtelten Bereiche gebildeten Baumes. Weiterhin wird innerhalb des Baumes eine Symboltabelle angelegt. Diese Tabelle ordnet den Bereichen die in ihnen deklarierten Bezeichner zu. Die Namensanwendungen werden in einer Kreuzreferenztabelle (*cross reference table*) abgelegt. Hier werden den Namensdeklarationen die zugehörigen Anwendungen zugeordnet. Die Attributierung muß für Programme einmalig vollständig berechnet werden.

4.2.6.2 Neuberechnung der Attributierung

Bei der erneuten Berechnung der Attribute nach Änderungen des AST soll allerdings eine vollständige Neuberechnung vermieden werden, da sich meist nur Teile des AST ändern. Da das MPS Änderungen des AST nur recht unspezifisch anzeigt (nur das Ersetzen oder Löschen von Teilbäumen kann festgestellt werden), kann auch die Feststellung, welche Attribute ungültig werden nur sehr global festgestellt werden. Da die Änderungen sehr komplex im Bezug auf die Attributierung sein können, z.B. das Einfügen oder Löschen von Bereichen, von Namensdeklarationen oder Namensanwendungen, müssen Kompromisse bei der Neuberechnung eingegangen werden. Es werden zwar doppelte Berechnungen durchgeführt, ein großer Teil der Attributierung kann in vielen Anwendungsfällen jedoch erhalten bleiben.

Zur Vermeidung mehrfacher Berechnungen wird die Berechnung der Attributierung bereichsweise vorgenommen. Für jeden Bereich soll die Bedingung gelten, das möglichst wenige Attribute von außerhalb auf diesen Bereich und keine Attribute auf innere Knoten dieses Bereichs verweisen. So wird die Attributierung des Bereichs gekapselt. Im Zusammenwirken mit einem Attribut, das anzeigt, ob innerhalb dieses Bereichs seit der letzten Berechnung Änderungen des AST durchgeführt wurden, kann man nun entscheiden, ob die Attributierung dieses Bereichs neu berechnet werden muß. Die nähere Beschreibung des Umfangs der Neuberechnungen erfolgt ausführlich in Abschnitt 4.2.6.7.

4.2.6.3 Namensräume

Bevor die Betrachtung der Attributierung im Rahmen des Baums der Bereiche, der Symboltabelle und der Kreuzreferenztabelle durchgeführt wird, erfolgt hier die Festlegung der Namensräume, die im folgenden verwendet werden:

(N1) Namensraum für die Namen des Programms, der Prozeduren, Module und Behandlungsroutinen, der Variablen, der Konstanten und der formalen Parameter
(N2) Namensraum für die Marken der Schleifen
(N3) Namensraum für die Namen von Ausnahmen

4.2.6.4 Bereichsbaum

Im folgenden wird zuerst die Identifikation der Konstrukte vorgenommen, die in ProSet Bereiche bilden. Anschließend wird die Attributierung für die Repräsentation des Bereichsbaumes festgelegt.

Identifikation der Bereiche

Folgende Programmeinheiten bilden Bereiche für die Namen des Namensraumes (N1). In Klammern ist die Bezeichnung der korrespondierenden Baumknoten angegeben:

(B1) Programmdefinition (*ProgDef*)
(B2) Prozedurdefinition (*ProcDef*)
(B3) Moduldefinition (*ModuleDef*)
(B4) Definition von Routinen zur Ausnahmebehandlung (*HandlerDef*)
(B5) Lambda-Definition (*Lambda*)

Zusätzlich zu diesen Programmeinheiten gibt es eine Reihe von Konstrukten, deren Verwendung eine implizite Deklaration von Bezeichnern verursachen. Dies sind diejenigen Konstrukte, die Iteratoren verwenden. Sie führen zusätzliche Bereiche ein, in denen die Iterationsvariablen implizit deklariert sind. Diese Bereiche erhalten allerdings eine andere Semantik bezüglich der Sichtbarkeitsregeln. Innerhalb eines solchen Bereiches sind die im umgebenden Bereich gemäß (B1) bis (B5) als lokal deklarierten Bezeichner weiterhin sichtbar.

Durch folgende Konstrukte werden Bereiche mit impliziten Deklarationen gebildet (in Klammern ist die Bezeichnung der Baumknoten angegeben). Auch diese Bereiche gelten für die Namen gemäß Namensraum (N1):

(I1) Tupel-Former und Mengen-Former mit Ausdrücken über Iteratoren (*TFormerIteration*, *SFormerIteration*), z.B. `{ [i, j]: i in S, j in T| p(i, j) }`
(I2) Quantifizierte Ausdrücke (*Quantifier*), z.B. `exists i in T| p(i)`
(I3) `for`-Schleifen (*ForStmtNoTrailer*)
(I4) `whilefound`-Schleifen (*WhileFoundStmtNoTrailer*)

Weitere Bereiche werden für den Namensraum der Marken von Schleifen (N2) durch die Schleifenkonstrukte gebildet (in Klammern folgt die Bezeichnung der Baumknoten):

(L1) `loop`-Schleifen (*LabLoopStmt*)
(L2) `repeat`-Schleifen (*LabRepeatStmt*)
(L3) `while`-Schleifen (*LabWhileStmt*)

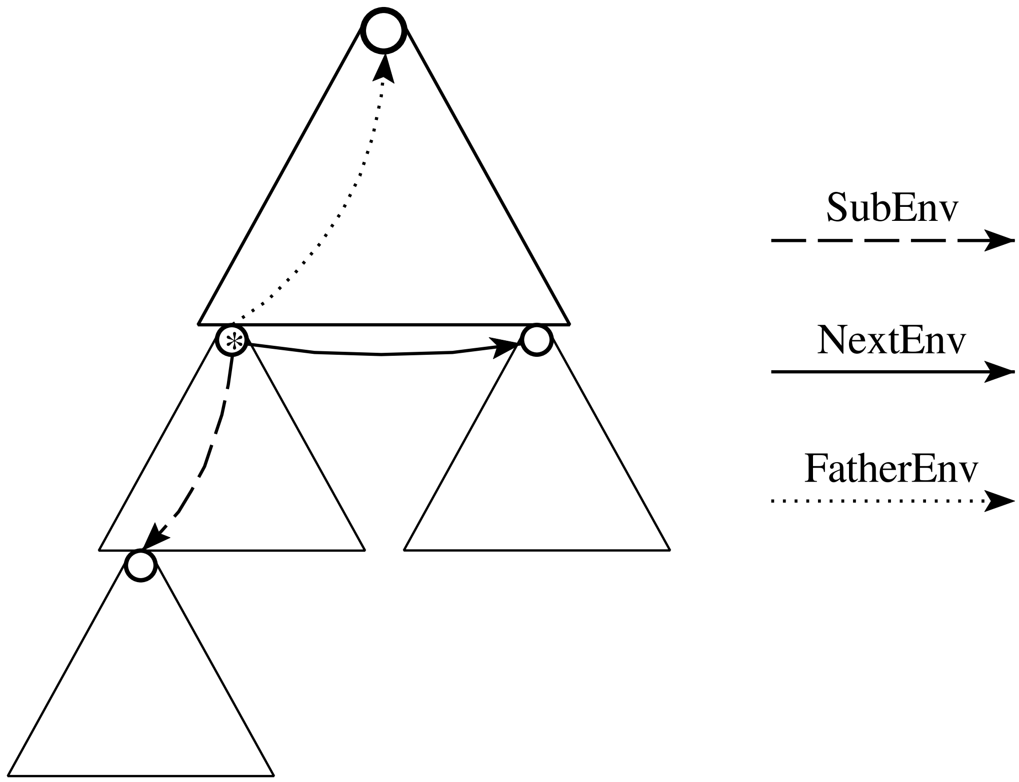


Abbildung 13 Attributierung für den Bereichsbaum

(L4) `for`-Schleifen (*LabForStmt*)
(L5) `whilefound`-Schleifen (*LabWhileFoundStmt*)

Wie bereits in Abschnitt 2.3.1.11 angesprochen wurde, ist der Namensraum für Ausnahmen (N3) flach und es werden somit keine Bereiche identifiziert.

Attributierung der Bereiche

Basis für die spätere Analyse bildet die Repräsentation des durch die Bereiche festgelegten Baumes durch Attributierung. Hierbei werden Verweise von einem Bereich zu dessen Vaterbereich, zu dem ersten Sohnbereich, sowie zu seinem Bruderbereich verwaltet. Der Ordnungsbegriff sei hier definiert als die Reihenfolge, in denen die Bereiche bei einem *preorder*-Baumdurchlauf erreicht werden.

Alle unter (B1) bis (B5) und (I1) bis (I4) aufgeführten AST-Knoten definieren folgende Attribute für die Verwaltung des Baums der Bereiche:

- Verweis auf den Vaterbereich: *FatherEnv*
- Verweis auf den ersten Sohnbereich: *SubEnv*
- Verweis auf den nächsten Bruderbereich: *NextEnv*

Die Attribute *NextEnv* bilden eine Liste von Sohnbereichen. Der erste dieser Sohnbereiche wird über das Attribut *SubEnv* an den Vaterbereich gebunden.

Im Bezug auf die in Abschnitt 4.2.6.2 geforderte Kapselung der Bereiche kann für die Attribute des Bereichsbaumes festgestellt werden, das ein Bereich von höchstens einem Attribut referenziert

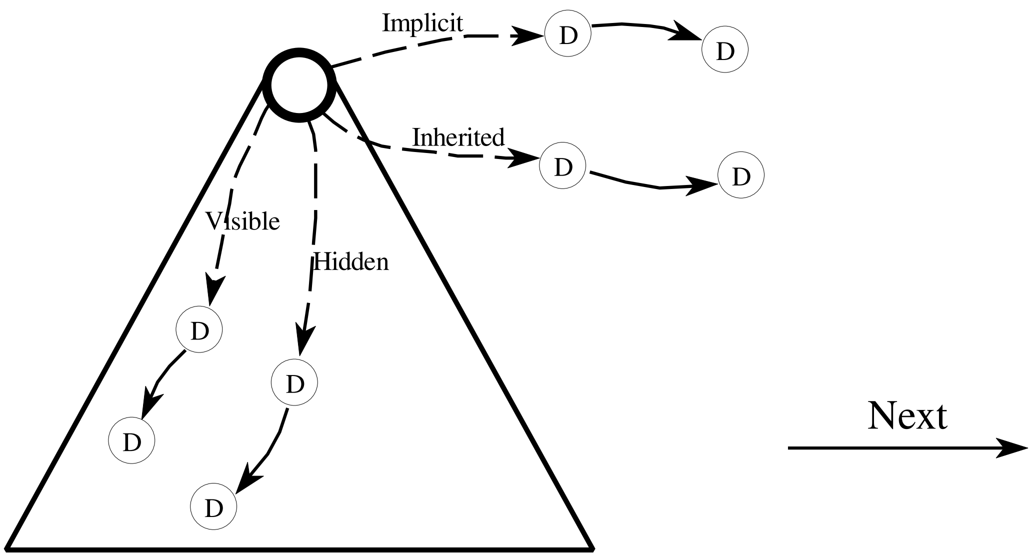


Abbildung 14 Symboltabelle eines Bereiches

wird. Dies ist entweder das Attribut *SubEnv*, falls dieser Bereich erster Sohnbereich ist, oder das Attribut *NextEnv*, falls er nicht erster Sohnbereich ist.

Abbildung 13 stellt die Attributierung der Bereiche dar. Jedes Dreieck symbolisiert einen Bereich. An der Grundseite der Dreiecke können sich weitere Dreiecke anschließen. Diese Hierarchie der Dreiecke entspricht der Schachtelung der Bereiche. In der Abbildung ist nur für den mit "*" gekennzeichneten Bereich die Attributierung aufgeführt. Von diesem Bereich ausgehend werden Attributverweise auf den ersten Sohnbereich, den nächsten Bruderbereich und den Vaterbereich, symbolisiert durch die unterschiedlichen Pfeile, verwaltet.

4.2.6.5 Symboltabelle

Die Symboltabelle wird bereichsweise organisiert. Dazu werden an jedem der in Abschnitt 4.2.6.4 unter (B1) bis (B5) und (I1) bis (I4) definierten Bereiche Attribute für die enthaltenen Namensdeklarationen bereitgestellt. Die Namensdeklarationen werden in Form von Listen miteinander verbunden. Um die unterschiedliche Art der Deklaration der Bezeichner hervorzuheben, werden global und lokal deklarierte Bezeichner in getrennten Listen verwaltet. Auch für implizit deklarierte Bezeichner wird eine eigenständige Liste bereitgestellt. Da implizite Deklarationen im AST nur in Form von Namensanwendungsknoten auftreten, wird für jede implizite Deklaration ein zusätzlicher Deklarationsknoten erzeugt, um die einheitliche Behandlung von expliziten und impliziten Deklarationen zu gewährleisten. Ebenso wird für Namensanwendungen eines nicht im gleichen Bereich deklarierten Bezeichners eine eigene Liste von zusätzlich erzeugten Deklarationsknoten angelegt.

Folgende Attribute werden für die Realisierung der Symboltabelle an den Knoten der Bereiche verwendet:

- Verweis auf den ersten global deklarierten Namen: *Visible*
- Verweis auf den ersten lokal deklarierten Namen: *Hidden*
- Verweis auf den ersten implizit deklarierten Namen: *Implicit*
- Verweis auf den ersten aus einem umgebenden Bereich verwendeten Namen: *Inherited*

Die Verkettung der Namensdeklarationsknoten (*IdDecl*) eines Bereiches erfolgt jeweils über Attribute mit dem Namen *Next*.

Abbildung 14 zeigt die Repräsentation der Symboltabelle eines Bereiches. Der durch das Dreieck symbolisierte Bereich enthält acht Namensdeklarationen. Diese werden durch mit "D" beschriftete Kreise dargestellt. Je zwei dieser Deklarationen sind explizit als global bzw. lokal deklariert, so daß die mit den Attributen *Visible* und *Next* bzw. *Hidden* und *Next* gebildeten Listen von Deklarationsknoten je zwei Elemente enthalten. Die Abbildung enthält weiterhin zwei implizit deklarierte Bezeichner und zwei aus einem umgebenden Bereich verwendete Bezeichner. Die zugehörigen erzeugten Deklarationsknoten sind in Listen mit den Attributverweisen *Implicit* und *Next* bzw. *Inherited* und *Next* an den Bereich gebunden.

Im Hinblick auf die Kapselung der Symboltabelle in jedem Bereich (vergleiche Abschnitt 4.2.6.2), wird für die in diesem Bereich angewendeten globalen Bezeichner aus umschließenden Bereichen eine Kopie in diesem und allen Bereichen zwischen Deklaration und Anwendung angelegt. Die globale Deklaration des Symbols und die Kopien werden über folgende Attribute der Knoten *IdDecl* miteinander verbunden:

- *FirstInherited* verweist auf die erste Kopie einer Namensdeklaration eines Sohnbereiches.
- *NextInherited* bildet die Verkettung der Kopien untereinander ab. Dieses Attribut verweist auf die nächste Kopie dieses Bezeichners in einem Bruderbereich.
- *VisibleDecl* verweist auf den Ursprung dieser Kopie in dem Vaterbereich.

Abbildung 15 stellt die Attributierung der Kopien von Namensdeklarationen dar. Einige Bereiche sind zum Zweck der Referenzierung numeriert. Die Abbildung enthält zusätzlich zu Deklarationsknoten auch zwei Namensanwendungen, die durch mit "A" bezeichnete Kreise repräsentiert werden (die Attributierung der Namensanwendungen wird im folgenden Abschnitt definiert). Diese Namensanwendungen korrespondieren zu der hervorgehobenen (globalen) Deklaration im mit 1 bezeichneten Bereich. Um die bereichsweise Kapselung der Attributierung zu unterstützen, ist

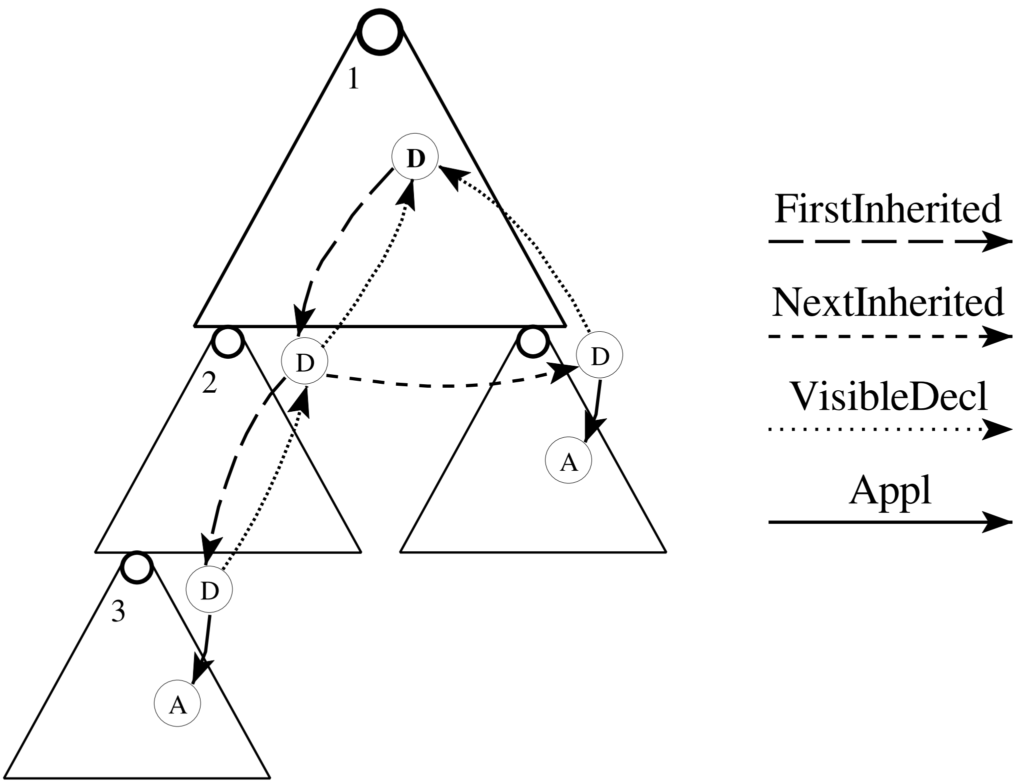


Abbildung 15 Attributierung der Benutzung globaler Bezeichner

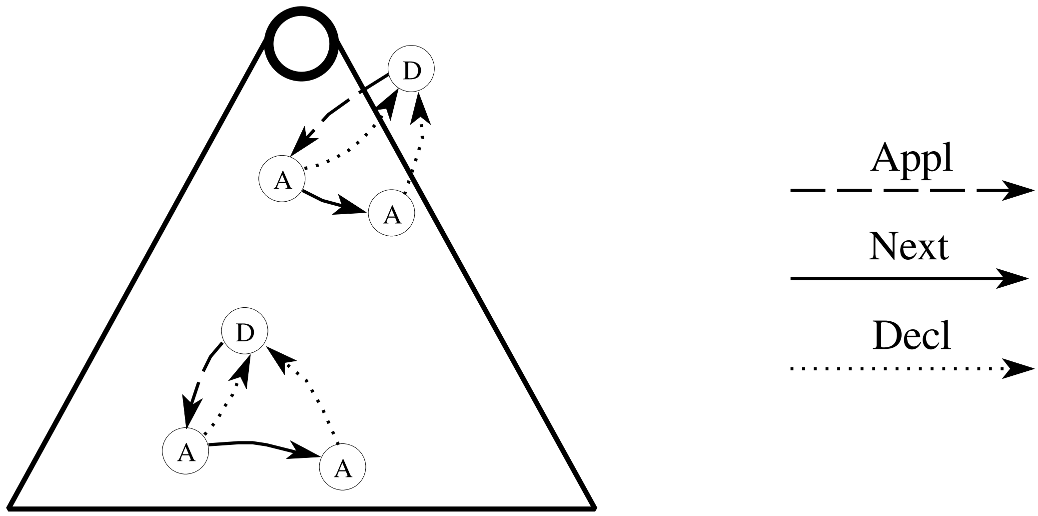


Abbildung 16 Kreuzreferenztabelle eines Bereiches

z.B. für die Namensanwendung in Bereich 3 eine Deklarationskopie angelegt worden, ebenso ist im zwischen Bereich 1 und 3 liegenden Bereich 2 eine Deklarationskopie angelegt worden. Die Deklarationsknoten sind über die Attributverweise *FirstInherited*, *NextInherited* und *VisibleDecl* verbunden.

Die auf diese Art definierte Attributierung für die Verwaltung nicht lokaler Deklarationen gewährleistet, daß auf jede Kopie eines globalen Bezeichners nur ein Verweis aus einem Vater- oder Bruderbereich existiert.

4.2.6.6 Kreuzreferenztabelle

Die Kreuzreferenztabelle erweitert die Attributierung um die in einem Bereich vorkommenden Namensanwendungen. Die Kreuzreferenztabelle ordnet den Namensdeklarationen des Bereichs die zugehörigen Namensanwendungen zu.

Sie wird repräsentiert durch das Attribut *Appl* an Namensdeklarationen, das auf die erste Anwendung dieser Namensdeklaration verweist. Die Attribute *Decl* der Namensanwendungen bilden wiederum einen Verweis auf die zugehörige Namensdeklaration. Die Verkettung der Namensanwendungen zu einer Deklaration erfolgt jeweils über das Attribut *Next* der Namensanwendung. Diese Art der Darstellung gilt sowohl für den Namensraum (N1) als auch für den Namensraum (N2).

Abbildung 16 zeigt die Attributierung der Kreuzreferenztabelle für den Namensraum (N1). Sie enthält vier Namensanwendungen, die durch mit “A” bezeichnete Kreise dargestellt werden. Zwei dieser Namensanwendungen korrespondieren zu einer expliziten Deklaration dieses Bereiches. Die Anwendungen dieser Deklaration sind durch die Attributverweise *Next* in Form einer Liste untereinander verbunden. Zusätzlich unterhält jede Namensanwendung einen Attributverweis *Decl* auf die zugehörige Deklaration. Die Deklaration wiederum verwaltet das Attribut *Appl* auf die erste Anwendung dieses Namens. Zwei weitere Anwendungen korrespondieren zu einer impliziten oder globalen Deklaration von außerhalb dieses Bereiches. Die Attributierung für diese Knoten ist analog zu der vorher beschriebenen.

Durch die Erweiterung der Symboltabelle um Deklarationsknoten für die Verwendung von globalen Symbolen umschließender Bereiche sind alle Namensanwendungen innerhalb eines Bereiches auch an Deklarationsknoten desselben Bereiches gebunden. Im Sinne der Kapselung der Attributierung von Bereichen sind keine Verweise auf Namensanwendungen aus anderen Bereichen vorhanden.

4.2.6.7 Umfang der Neuberechnung

Während in Abschnitt 4.2.6.2 die Neuberechnung der Attributierung schon motiviert wurde, blieben dort noch der Umfang und weitere Nebenbedingungen ungenannt.

Aufgrund der unspezifischen Betrachtung in Bezug auf die Art der im AST durchgeführten Modifikationen, wurde der Ansatz gewählt, vollständige Bereiche neu zu berechnen. Die Feststellung, welche Bereiche modifiziert wurden, wird mit Hilfe des Attributes *Uptodate* getroffen. Da innerhalb eines modifizierten Bereiches beliebige Änderungen stattgefunden haben können, ist die Korrektheit der Attributierung für diesen Bereich selbst, sowie für dessen Sohnbereiche nicht mehr gewährleistet, wenn nach einer Modifizierung das Attribut *Uptodate* zurückgesetzt wurde. So kann sich z.B. die Attributierung der Söhne ändern, wenn Deklarationen in den Vater eingefügt oder gelöscht wurden. Aus diesem Grund wird auch die Attributierung aller Sohnbereiche neu berechnet. Die Neuberechnung beginnt also bei den äußersten geänderten Bereichsknoten des AST.

Bevor die Neuberechnung der Attributierung stattfinden kann, müssen alle Bezüge eines äußersten modifizierten Bereiches von außerhalb bereinigt werden:

- Alle Kopien von Namensdeklarationen in der durch das Attribut *Inherited* bezeichneten Liste müssen aus den Listen der Originaldeklarationen gelöscht werden. Die Originaldeklaration ist durch Verwendung des Attributes *VisibleDecl* erreichbar.
- Der Bereich selbst wird aus dem Bereichsbaum entfernt. Hierzu ist dieser Bereich aus der Liste der Sohnbereiche seines Vaterbereiches zu entfernen. Der Vaterbereich wird durch das Attribut *FatherEnv* bezeichnet.
- Alle Attribute innerhalb des modifizierten Teilbaumes müssen gelöscht werden.

Nach diesen Vorarbeiten ist die Attributierung des gesamten AST korrekt, aber unvollständig. Durch die erneute Berechnung der Attributierung für die geänderten Bereiche wird die Attributierung des AST vervollständigt.

4.2.6.8 Kontextsensitive Einschränkungen

Im folgenden wird die Überprüfung der in Abschnitt 3.3.4.1 beschriebenen Einschränkungen kurz skizziert. Diese werden im wesentlichen durch Analyse innerhalb des Syntaxbaumes oder durch Analyse von Symboltabelle und Kreuzreferenztabelle, also durch Zugriff auf Informationen, die in Attributen gespeichert sind, durchgeführt. Die Analyse des Syntaxbaumes ist jeweils so ausgelegt, daß nur Knoten durchlaufen werden, die einen geringen Abstand voneinander haben. Der Abstand von zwei Knoten sei hier definiert als die kleinste zu durchlaufende Anzahl von Verweisen zu Vater- oder Sohnknoten oder Attributverweisen.

Namensanalyse

Die Überprüfung der Identität der Namen in der Kopfzeile und der Fußzeile der Programmeinheiten Programm, Prozedur, Behandlungsroutine, Modul und der Schleifen mit Marken erfolgt durch Analyse des Syntaxbaumes. Das Auffinden von Doppeldeklarationen innerhalb eines Bereiches wird unter Benutzung der Symboltabelle durchgeführt. Hierbei werden die Tabelleneinträge für diesen Bereich untersucht.

Allgemeine Kontextüberprüfungen

Die Überprüfung von Anweisungs- und Ausdruckskontexten (vergleiche Abschnitt 3.3.4.1) erfolgt bereichsweise durch Überprüfung der umschließenden Konstrukte. Der zu ermittelnde Kontext der Anweisungskonstrukte `return`, `resume`, `return commit` und `self` ist der umschließende Bereichsknoten gemäß (B1) bis (B5).

Für `quit` und `continue` wird das umgebende Schleifenkonstrukt durch Aufstieg im AST ermittelt. Wird während der Analyse der umgebenden Konstrukte ein Bereichsknoten gemäß (B1) bis (B5) erreicht, so wurde ein falscher Kontext ermittelt.

Für die Kontextbeschränkung der Ausdrücke mit `into` wird die umgebende Anweisung ermittelt. Handelt es sich hierbei nicht um eine `meet`-Anweisung, so ist `into` in einem falschen Kontext verwendet worden. Für die Verwendung von `$` innerhalb von Ausdrücken ist der umgebende Ausdruck zu ermitteln. Dieser muß in einen Ausdruck mit `into` oder in die Bedingung eines Tupelraum-Musters eingebettet sein.

Für die Analyse der Verwendung von Prozedur-, Modul- und Behandlungsroutinennamen wird der verwendete Kontext bestimmt. Dies ist jeweils das umschließende Konstrukt. Für den entsprechenden Knoten kann sofort entschieden werden, ob er gültig gemäß der Definition in Abschnitt 3.3.4.1 ist.

Weitere Kontextüberprüfungen betreffen die in `closure`-Anweisungen erlaubten Namen. Zu den dort verwendeten Namen werden die korrespondierenden Deklarationen geprüft. Handelt es sich bei dem verwendeten Namen weder um einen Prozedur- noch um einen Modulnamen, so wird ein Kontextfehler erkannt.

Die Analyse von Assoziationen für Behandlungsroutinen wird auf die gleiche Art durchgeführt. Die Deklaration der verwendeten Namen wird ermittelt. Handelt es sich nicht um die Definition einer Behandlungsroutine, so ist dies ein Kontextfehler.

Für die Analyse der Auslösung vordefinierter Ausnahmen werden alle Konstrukte zur Auslösung von Ausnahmen analysiert (dies sind `escape`-, `signal`- und `notify`-Anweisungen). Für die dort verwendeten Ausnahmenamen wird ermittelt, ob diese vordefiniert sind. Handelt es sich um eine vordefinierte Ausnahme, so wird ermittelt, ob die Parameterliste die entsprechende Anzahl an Parametern enthält. Die in der Sprachdefinition [DFG+92a] aufgeführten vordefinierten Ausnahmen verwenden keine Parameter. Die nachträglich eingeführte Ausnahme `p_missing_entry` verwendet einen Parameter.

Kontext von Prozeduraufrufen

Die in Abschnitt 3.3.4.1 geforderte Überprüfung der Anzahl und der Art der aktuellen Parameter von Prozeduraufrufen wird mit Hilfe der Kreuzreferenztabelle und durch Analyse des AST durchgeführt. Für die Feststellung, ob ein syntaktisches Konstrukt den Aufruf einer Prozedur beschreibt, ist die Deklaration des Namens aus der Kreuzreferenztabelle zu ermitteln und festzustellen, ob die Namensdeklaration eine Prozedurdefinition beschreibt. Handelt es sich um einen Prozeduraufruf, so können durch Analyse des Syntaxbaumes die Parameteranzahlen innerhalb der Listen der formalen und der aktuellen Parameter verglichen werden.

Die Sicherstellung, daß ein aktueller Parameter einen gültigen l-Wert (siehe Abschnitt 2.3.1) beschreibt, falls er als `wr`- oder `rw`-Parameter verwendet wird, ist durch eine Baumanalyse sicherzustellen. Hier muß für die im AST verwendeten syntaktischen Ausdruckstypen rekursiv die Beschränkung auf eine Teilmenge der für l-Werte gültigen Alternativen sichergestellt werden.

Sonstige Kontextüberprüfungen

Alle in Abschnitt 3.3.4.1 geforderten sonstigen Einschränkungen werden durch Analyse des Syntaxbaumes sichergestellt. Für Deklarationen (nicht Persistenzdeklarationen) mit dem Deklarationsattribut `constant` muß festgestellt werden, ob alle deklarierten Namen initialisiert werden. Dies ist der Fall, wenn der Deklarationsknoten in einen Teilbaum des Typs *SingleVarAssigned* eingebettet ist.

Weiterhin wird für konstant deklarierte Bezeichner untersucht, ob sie als l-Wert verwendet werden. Ist dies der Fall, so erfolgt eine Fehlermeldung. Zusätzlich wird überprüft, ob dieser Name in der Parameterliste eines Prozeduraufrufs verwendet wird. Ist der korrespondierende formale Parameter als `wr` oder `rw` definiert, so wird ebenfalls ein Fehler erkannt.

Die in Modulen deklarierten persistenten Bezeichner werden dahingehend überprüft, ob als Deklarationsart `visible persistent` verwendet wird. Ist dies der Fall, so erfolgt eine Fehlermeldung.

Arbeitsweise und Benutzerinteraktion

Auf Anforderung des Benutzers werden die oben beschriebenen Überprüfungen durchgeführt. Enthält das Programm keine Fehler, so wird der Erfolg dem Benutzer in Form eines Statusfensters angezeigt und die Attributierung des Programms wurde berechnet. Fehler werden dem Benutzer in Form eines Listenfensters von Fehlermeldungen angezeigt. Bei Selektion eines der Fehlereinträge wird das fehlerhafte Programmkonstrukt im Editor markiert, wobei unter Umständen der angezeigte Programmausschnitt geändert wird. Der Benutzer kann nach Beseitigung der Fehler weitere Überprüfungen durchführen. Abbildung 23 in Anhang A zeigt einen Bildschirmabzug des Fehlerfensters.

4.2.6.9 Weitere Analysefunktionen

Aufbauend auf Symboltabelle und Kreuzreferenztabelle werden folgende Analysefunktionen für den Benutzer angeboten.

Ausgabe der Symboltabelle

Die Informationen der Symboltabelle für einen Bereich werden in einem Fenster in Listenform ausgegeben. Es werden die in der Symboltabelle dieses Bereiches enthaltenen Bezeichner gruppiert nach globalen, lokalen, impliziten und nicht-lokal verwendeten globalen Deklarationen angezeigt. Zusätzlich wird die Art der Deklaration in Form einer Liste von Merkmalen (Prozedur, Modul, Behandlungsroutine, Konstante, Persistenzeigenschaft) angezeigt.

Suche der Deklarationsstelle zu einem Bezeichner (*use-definition-chain*)

Zu einer markierten Namensanwendung wird die zugehörige Deklarationsstelle gesucht. Falls es sich bei der Anwendung um einen implizit deklarierten Bezeichner handelt, kann keine Deklaration angezeigt werden. Dann soll das erste angewandte Auftreten des Bezeichners markiert werden.

Suche aller Anwendungen eines Bezeichners (*definition-use-chain*)

Von der markierten Namensdeklaration ausgehend kann die erste Anwendung dieses Namens aufgesucht werden. Von jeder Namensanwendung ausgehend kann dann jeweils die nächste Anwendung aufgesucht werden. Dies entspricht der Navigation entlang der Verbindungen von Namensanwendungen des Bezeichners.

Abgrenzung

Die Überprüfung der in Abschnitt 3.3.4.2 beschriebenen Qualitätskriterien wird im Rahmen dieser Arbeit nicht vorgenommen. Dies würde eine teilweise umfangreiche Kontrollflußanalyse erfordern.

4.2.7 Anbindung weiterer Werkzeuge

Für die Nutzung von Diensten externer Werkzeuge, aber auch die Bereitstellung von Diensten seitens der PU wurde in Abschnitt 3.2.3 die Bereitstellung einer Kommunikationskomponente gefordert. Neben dem direkten Aufruf externer Werkzeuge, der zwar einfach bereitzustellen ist, aber lange Ladezeiten verursacht, bietet sich eine flexiblere Möglichkeit an: Die Bereitstellung von Kommunikationskanälen zu externen Werkzeugen führt zu einer losen Kopplung der Umgebung und dieser Werkzeuge.

Für die Kommunikation sind Standardsysteme verfügbar. Sie bieten einerseits Flexibilität bei Laden und Beenden von Werkzeugen. Beispielsweise können komplexe Werkzeuge bei der ersten Benutzung geladen werden, bei folgenden Aufrufen kann dann dasselbe Werkzeug verwendet werden.

Die PU kann Nachrichten an externe Werkzeuge versenden und damit die Dienste dieser Werkzeuge nutzen. Der umgekehrte Kommunikationsweg, das Empfangen von Nachrichten seitens der PU könnte nur realisiert werden, wenn der interne Aufbau der PU und somit das zugrundeliegende BETA-Rahmenwerk dieses antizipieren würde. Hier wäre ein Eingriff in die Ablaufsteuerung des BETA-Rahmenwerks nötig, der zur Zeit nicht vorgesehen ist. Eine für das BETA-Rahmenwerk beispielhaft von Mjølner realisierte Kopplung mit dem Mjølner-Entwurfswerkzeug basiert auf einem manuell realisierten, unflexiblen Kommunikationsmechanismus.

Für die Kommunikation mit anderen Werkzeugen wird das Nachrichtenvermittlungssystem ToolTalk der Firma Sun Microsystems [Sun91] verwendet. Dieses ermöglicht den Austausch von Nachrichten zwischen unterschiedlichen Prozessen. Für das Senden einer Nachricht muß dem Sender der Empfänger nicht unbedingt bekannt sein. Der Sender übergibt eine zu versendende Nachricht an den ToolTalk-Dienst. Dieser ermittelt je nach Typ der Nachricht einen angemeldeten Empfänger, oder startet einen Empfänger neu, falls dies in der Konfiguration berücksichtigt wurde. Die Nachricht wird dem ermittelten Empfänger dann zugestellt.

Für die Generierung und Absetzung von Nachrichten an externe Werkzeuge werden in der Prototyping-Umgebung entsprechende Aufrufe eingebunden. Dabei muß für jeden Aufruf eines externen Werkzeuges der Nachrichtentyp und die zu übertragenden Parameter ermittelt werden. Der Nachrichtenaustausch mit dem ToolTalk-Dienst erfolgt durch die Kommunikation mit einem Sitzungsprozeß (*ttsession*) über eine Programmierschnittstelle (*libtt*).

Die Kommunikation mit externen Werkzeugen wird beispielhaft für die PROSET-Persistenzwerkzeuge entwickelt. Dazu werden entsprechende Nachrichten mit dem anschließend genannten Aufbau verwendet:

- *p-file tool*: Das Werkzeug wird verwendet, um *p-files* für persistente PROSET-Werte anzulegen oder zu löschen. Es wird aus der PU heraus aufgerufen, um *p-files* anzulegen. Hierzu wird der Name des *p-files* aus der Persistenzdeklaration ermittelt, die Verbindung zu dem ToolTalk-Sitzungsprozeß wird hergestellt, die Nachricht für das Werkzeug generiert und an den ToolTalk-Dienst übergeben.
- *p-file editor*: Mit Hilfe dieses Editors können persistente PROSET-Werte angezeigt oder manipuliert werden. In der Prototyping-Umgebung wird dieses Werkzeug verwendet, um Werte zu modifizieren oder zu betrachten. Für diesen Zweck wird ein Nachrichtentyp verwendet, der den Editor veranlaßt, das Fenster für Anzeige und Editierung eines persistenten Wertes zu öffnen. Die zu generierende Nachricht beinhaltet den Bezeichner der persistenten Deklaration und den Namen des zugeordneten *p-files*. Diese Informationen werden aus der Persistenzdeklaration innerhalb des AST ermittelt.

Zusätzlich zu der Einbindung der Persistenzwerkzeuge durch Nachrichtenübermittlung kann der vorhandene PROSET-Übersetzer aus der Umgebung heraus aufgerufen werden, um das aktuell bearbeitete Programm zu übersetzen. Der Übersetzer wird durch direkten Aufruf angebunden. Der Übersetzer kann durch Auswahl eines Menüeintrages aufgerufen werden, wenn zuvor die semantische Korrektheit des Programms überprüft wurde und das Programm vollständig ist. Dann ist eine fehlerfreie Übersetzung des Programms möglich. Über einen zusätzlichen Menüeintrag können Parameter für die Übersetzung angegeben werden, die bei Aufruf zusätzlich zu dem Programmnamen an den Übersetzer übergeben werden.

5 Realisierung

In diesem Kapitel werden die Module der PU und das Vorgehen beim Test der einzelnen Module beschrieben. Hier werden auch Kritikpunkte an dem verwendeten BETA-Metaprogrammiersystem aufgezeigt.

Für die Unterstützung bei der Entwicklung stand der *support* von Mjølner-Informatics in Dänemark per E-mail als Ansprechpartner zur Verfügung. Im Verlauf der Realisierung mußte diese Unterstützung des öfteren in Anspruch genommen werden. Um die Kommunikation mit Mjølner zu erleichtern, erfolgte die Namensgebung von Bezeichnern und die Kommentierung der Quelltexte in englischer Sprache.

5.1 Modulbeschreibung

Die PU besteht aus einer Anzahl von Modulen, deren Abhängigkeiten in Abbildung 17 (verfeinerte Architektur) dargestellt sind. Die Pfeile bezeichnen dabei “benutzt”-Beziehungen der einzelnen Module, wobei dicke Pfeile eine Bündelung von “benutzt”-Beziehungen zwischen mehreren Modulen einzelner Teilsysteme symbolisieren. Bevor in den folgenden Abschnitten diese Module näher erläutert werden, folgt hier zuerst die Beschreibung der einzelnen Teilsysteme:

Die **PROSET-Sprachbeschreibung und Unparsing-Beschreibung** beschreibt die Syntax für die Definition der ASTs und für den Parser (implizit auch die Auswahlmöglichkeiten innerhalb des Editors) sowie die konkrete Syntax von PROSET in Form von Unparsing-Schemata.

Das Modul **Statische Analyse** realisiert die statischen Überprüfungen, sowie die Analysefunktionen. Die weitere Unterteilung wird in Abschnitt 5.1.2 beschrieben.

Das **ToolTalk-Subsystem** realisiert die Nutzung der Dienste der Persistenzwerkzeuge über den Nachrichtendienst ToolTalk.

Das Teilsystem **UI/Ablauf PROSET** implementiert die Erweiterung des Systemmenüs und der zusätzlichen Kontextmenüs. Außerdem werden hier zusätzliche Dialogfenster und weitere Dialogelemente bereitgestellt. Hier erfolgt auch die Einbindung der PROSET-spezifischen Werkzeuge in die Ablaufsteuerung des Rahmenwerks.

5.1.1 PROSET-Sprachbeschreibung für das Metaprogrammiersystem

Für die Verwendung des MPS [Mjøl94b] und des BETA-Rahmenwerks ist die Erstellung einer Sprachspezifikation und einer Unparsing-Beschreibung für PROSET erforderlich.

Die Spezifikation der AST-Definition und die Syntaxbeschreibung für den Parser erfolgt in der Datei `proset-meta.gram`. Die Beschreibung der Unparsing-Schemata erfolgt in der Datei `proset-pretty.pgram`.

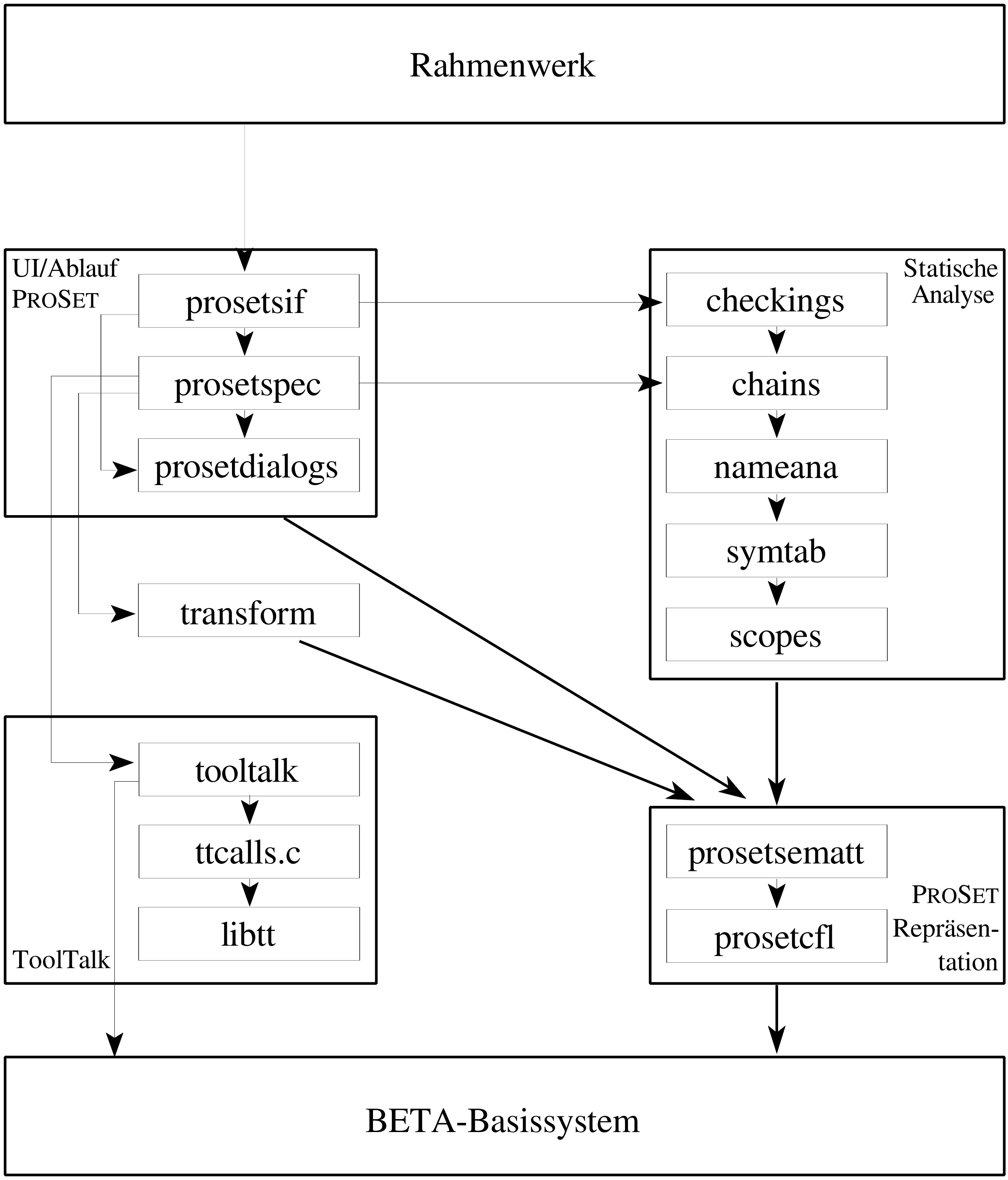


Abbildung 17 Verfeinerte Architektur der Prototyping-Umgebung

Die entstandene Spezifikation für PROSET ist insgesamt recht umfangreich und konnte nicht mit den Standardwerkzeugen des MPS verarbeitet werden. Für die Parsererstellung wurde ein spezieller Parsergenerator mit vergrößerten Tabellen benötigt. Ebenso wurde eine spezielle Version des Werkzeuges zur Erstellung der BETA-Schnittstelle zu PROSET erforderlich, das alle von PROSET verwendeten Sonderzeichen berücksichtigte.

Konventionen

Im folgenden sollen nun kurz die Konventionen bei der Namensgebung der AST-Knoten aufgeführt werden. Für Folgen gleicher Inkremente wurde als Repräsentationsmittel die vom MPS angebotenen Listenproduktionen verwendet. Diese Produktionen erhalten als Namenssuffix *List*. Optionale syntaktische Einheiten werden durch das Präfix *Opt* kenntlich gemacht. Da die Namen der Nichtterminale durch den Editor innerhalb der Expandierungsmenüs verwendet werden, ist auf eine möglichst aussagefähige Namensgebung Wert gelegt worden.

Eine weitere Konvention ergibt sich aus einem fehlerhaften Verhalten des Unparsers bei leeren Listen: Die Repräsentation von Listen erfolgt ausschließlich durch Listenproduktionen mit mindestens einem Element. Um Listen ohne Elemente repräsentieren zu können werden zusätzliche Optionsproduktionen verwendet.

Namensräume

Die Repräsentation unterschiedlicher Namensräume erfolgt über verschiedene Nichtterminalpaare, die auf die Basisnichtterminale des MPS (*NameDecl* für Deklarationen und *NameAppl* für Anwendungen) abgeleitet werden. So werden für Namen des Namensraumes (N1) das Nichtterminal *IdDecl* für Deklarationen und *Id* für Anwendungen verwendet. Für den Namensraum (N2) lauten die Nichtterminale für Deklarationen und Anwendungen *LabelDecl* und *LabelId*. Die verschiedenen nicht analysierten Namen, wie die Namensanwendungen von Modulprozeduren in der Importliste von Instanziierungen und bei Aufruf von Modulprozeduren werden durch das Nichtterminal *ModuleProcId* realisiert. Die Ausnahmenamen des Namensraumes (N3) werden durch die Verwendung des Nichtterminals *ExcId* realisiert.

Die Struktur der abstrakten Syntaxbäume, insbesondere die Auswahlmöglichkeiten des Editors werden genauer im Benutzerhandbuch (Anhang A) beschrieben. Dort sind für die wichtigsten Auswahlmöglichkeiten Kontextmenüs und deren Abhängigkeiten angegeben.

Bei der Realisierung der Unparsing-Schemata kam neben der Starrheit des adaptiven Unparsing-Algorithmus des MPS noch ein weiterer Problempunkt zum Tragen. Falls das letzte Element in einer Produktion optional ist, und die leere Expandierung gewählt wurde, so wird der vorhergehende Zwischenraum trotzdem ausgegeben, so daß beispielsweise bei dem Unparsing von Anweisungen ohne Assoziation von Behandlungsroutinen ein unerwünschter Abstand zu dem die Anweisung abschließenden Semikolon entsteht.

5.1.2 Modul statische Analyse

Das Modul zur Realisierung der statischen Analyse besteht aus weiteren Modulen:

`scopes.bet` realisiert die Erstellung und Handhabung des Bereichsbaumes und weitere Funktionalität zur Handhabung von Bereichsknoten.

`symtab.bet` realisiert die Erstellung und Verwaltung der Symboltabelle.

`nameana.bet` realisiert die Erstellung der Kreuzreferenztabelle.

`chains.bet` realisiert Zugriffe auf die Kreuzreferenztabelle für die Verwendung der *definition-use-chain* und der *use-definition-chain*.

`checkings.bet` enthält die Funktionalität zur Überprüfung der semantischen Einschränkungen.

Weitere Basisfunktionalität wird von folgenden Modulen zur Verfügung gestellt:

`debugg.bet` enthält einfache Muster zur ein- und ausschaltbaren Ausgabe von Meldungen zur Unterstützung der Fehlersuche.

`prosetsematt.bet` enthält Basismuster für verschiedene Baumoperationen, wie das Auffinden des zu einer Namensdeklaration gehörigen Konstruktes, Kontrollmuster für den Test auf Expandiertheit, Zugriffsmuster für komfortablen Zugriff auf Attribute von Knoten über Namen statt über deren Indexnummern (*AttributeInterface*), Muster für den Zugriff auf eine durch Attribute verkettete Liste von Knoten (*ListInterface*), allgemeine, erweiterbare Muster für den Auf- und Abstieg innerhalb des AST und weitere Muster für die Erzeugung von Kontrollausgaben zur Fehlersuche.

Für die Erweiterung der durch das MPS bereitgestellten Basismuster (z.B. *Ast*, *Cons* und *Expanded*) durch werkzeugabhängige Muster und die Erweiterung der Muster der generierten Schnittstelle zu ProSet sieht das MPS die Verwendung des Fragmentsystems vor. Das Fragmentsystem gestattet ausschließlich das Hinzufügen von nicht-virtuellen Musterdeklarationen zu den bereits vorhandenen. Dies bedeutet, daß die Verwendung von virtuellen Mustern oder Referenzen (Instanzen von Mustern) ausgeschlossen ist. Insbesondere muß durch den Verzicht auf virtuelle Muster auf ein wichtiges Ausdrucksmittel der objektorientierten Sprache BETA verzichtet werden. Dies führt in einigen Fällen zu einer umständlichen Realisierung und zu vielen Fallunterscheidungen.

Bei der Navigation durch die ASTs zeigt sich eine unschöne Handhabung bei der Verarbeitung unvollständiger Programme: Das Ab- und Aufsteigen muß mit BETA-Referenzen des allgemeinsten Typs (*Ast*) erfolgen, da beim Sohnzugriff entweder Referenzen auf nicht-expandierte Knoten (*Unexpanded*) oder auf expandierte Knoten (Submuster von *Cons* oder *List*) zurückgegeben werden. Diese sind nicht typkompatibel. Bei jeder abzusteigenden Stufe des Baumes muß die erhaltene Referenz auf Expandiertheit getestet werden, um durch eine anschließende Zuweisung an einen korrekt qualifizierten Bezeichner die Verwendung der speziellen Operationen des Knotens ermöglichen zu können. Dies führt zu recht komplexen und umständlichen Formulierungen bei Baumnavigationen, die einerseits eine hohe Fehleranfälligkeit verursachen, andererseits die Lesbarkeit erschweren. Eine Verwendung von deklarativen Ansätzen zur Beschreibung von Baumstrukturen wäre eine wünschenswerte Erweiterung des MPS.

5.1.2.1 Analyse der Bereiche

Bereiche werden durch verschiedenartige Knoten dargestellt. Damit die gemeinsame Funktionalität nicht mehrfach an allen Bereichsknoten definiert werden muß, wird sie an einer gemeinsamen Oberklasse der Muster für Bereiche definiert. Da die Spezifikation von ProSet keine gemeinsame Oberklasse für Bereiche in der Sprache definieren kann, wird die MPS-Klasse *Expanded* als gemeinsame Oberklasse verwendet. Innerhalb der hier definierten Muster wird über explizite Typüberprüfungen sichergestellt, daß diese nur von Bereichsknoten verwendet werden. Der Bereichsbaum wird in einem Baumdurchlauf unter Benutzung der Listenabstraktion aus `prosetsematt.bet` erstellt.

Als weitere Funktionalität sind hier Muster zum Attributzugriff für Bereichsknoten, Muster zum Auffinden des umschließenden Bereiches, zur Iterierung über die eingebetteten Bereiche, Iterierung über die umschließenden Bereiche und Muster für den Durchlauf eines Bereiches über alle oder bestimmte Knotenarten enthalten.

5.1.2.2 Symboltabelle und Namensanalyse

Die Erstellung der Symboltabelle und der Kreuzreferenztabelle wird in einem Durchlauf des Bereichsbaumes durchgeführt. Die Berechnung der Tabellen wird bereichsweise in zwei Durchläufen dieses Bereichs durchgeführt, in denen zuerst die Namensdeklarationen behandelt werden und anschließend die Namensanwendungen.

Die Symboltabelle wird dabei in zwei Schritten erstellt. Im ersten Schritt werden die Namensdeklarationsknoten *IdDecl* des Bereichs betrachtet. Diese werden anhand ihrer Deklarationsattribute in die entsprechende Liste für lokale oder global sichtbare Bezeichner eingefügt. Die Informationen zu impliziten Deklarationen und der Benutzung nicht-lokal deklarierter Bezeichner werden erst in der Namensanalyse verfügbar. Dort wird die Symboltabelle als zweiter Schritt entsprechend erweitert.

In der Namensanalyse werden in einem Durchlauf des Bereichs alle Namensanwendungen, das heißt alle Knoten vom Typ *Id* betrachtet. Zu diesen Knoten wird zuerst in der Symboltabelle desselben Bereichs, anschließend in den umschließenden Bereichen nach einer Deklaration gesucht. Wird keine Deklaration gefunden, so wird ein neuer Deklarationsknoten erzeugt und in die Liste für implizite Deklarationen der Symboltabelle dieses Bereichs eingefügt. Wird eine Deklaration in einem umschließenden Bereich gefunden, so wird ein Deklarationsknoten neu erzeugt und in die Liste für nicht-lokal verwendete Bezeichner (*Inherited*) eingefügt. Die Namensanwendungen und -deklarationen werden dann wie in Abschnitt 4.2.6.6 beschrieben miteinander verbunden.

Die Namensanalyse der Schleifen mit Marken erfolgt in einem Durchlauf des Bereiches, in dem die Namensanwendungen von Marken (*LabelId*) betrachtet werden. Zu diesen Namen wird durch Analyse des AST ein umschließendes Schleifenkonstrukt ermittelt, das diese Marke definiert. Die Anwendung des Markennamens und die Deklaration werden ebenso wie die Namen des Namensraumes (N1) miteinander verbunden (vergleiche Abschnitt 4.2.6.6).

Für alle weiteren Namen (Namen von Modulprozeduren in Aufrufen und Modulinstanziierungen, sowie Ausnahmenamen) wird keine Namensanalyse durchgeführt.

5.1.2.3 Überprüfung semantischer Einschränkungen

Die Einschränkungen werden durch Analyse des Syntaxbaumes und Verwendung der Kreuzreferenztabelle bereichsweise überprüft. Während eines Durchlaufs des Bereichs werden alle zu überprüfenden Knoten herausgefiltert. Die an diesen Knoten vorhandenen Muster zur Überprüfung der Einschränkung werden aufgerufen. Diese liefern als Ergebnis eine entsprechende Fehlernummer, falls die Überprüfung einen Fehler erbrachte. Der fehlerhafte Knoten wird dann mittels eines durch das MPS bereitgestellten Attributes markiert und mit der ermittelten Fehlernummer versehen. Die Fehlerattribute werden durch das Präsentationsmodul ausgelesen und dem Benutzer präsentiert.

5.1.3 Transformationsmodul

Die Realisierung der Transformationen wird als (Prozedur-)Muster des zu ersetzenden oder zu behandelnden Baumknotens realisiert. Diese Muster werden durch das Präsentationsmodul aufgerufen. Falls der aktuelle Baumknoten ersetzt wird, liefern die Transformationsmuster den ersetzten Knoten zurück.

5.1.4 Anbindung externer Werkzeuge

Die Anbindung externer Werkzeuge ist mittels ToolTalk realisiert worden. Der Aufbau der Nachrichten erfolgt in Funktionen der Sprache C. Diese C-Funktionen sind in die PU über die BETA-Bibliothek zur Einbindung externer Funktionen (*external*) eingebunden. Während die Datei `ttcalls.c` die C-Funktionen enthält, wurde deren Anbindung in der Datei `tooltalk.bet` vorgenommen. Für jede zu versendende Nachricht enthält die Datei `ttcalls.c` eine entsprechende Funktion, deren Parameter denjenigen der Nachricht entsprechen. Innerhalb der C-Funktionen wird dann der Aufbau der Nachricht vorgenommen und die Nachricht an den ToolTalk-Dienst übergeben.

5.1.5 Präsentationsmodul

Für die Realisierung der Oberfläche stand eine Vorabversion des BETA-Rahmenwerks (Sif Version 5.0) zur Verfügung. Diese Version wurde zeitgleich mit dem internen Test bei Mjølner-Informatics für die Erstellung der PU zur Verfügung gestellt. Die Möglichkeiten für die Erweiterung des Rahmenwerks wurden in Form eines rudimentären Musterprogramms demonstriert. Eine Dokumentation des Rahmenwerks existierte während der Erstellung dieser Arbeit nicht.

Diese Version des BETA-Rahmenwerks erlaubt nun erstmals die Realisierung der in dem Referenzhandbuch des MPS [Mjøl94b] dargestellten Erweiterungen. Für die Erstellung der Oberfläche des Rahmenwerks wurde von Mjølner eine plattformunabhängige Programmierschnittstelle verwendet.

Abgesehen von dem unvollständigen Umfang der Vorabversion (z.B. fehlende *online*-Hilfe) sind nur wenige Probleme mit dieser Vorabversion aufgetreten. Das wichtigste Problem tritt im Bereich des inkrementellen Unparsens auf. Hier kann es vorkommen, daß die Darstellung bei Änderungen im AST fehlerhaft ist. Dies äußert sich teilweise durch mehrfache Ausgabe der Knoten oder durch nicht aktualisierte Ausgaben. Die fehlerhaften Ausgaben können teilweise durch expliziten Aufruf des Unparsers (Menüpunkt *reprettyprint*) beseitigt werden. In einigen Fällen führt dieses fehlerhafte Verhalten des Unparsers allerdings zum Abbruch der Ausführung der Umgebung.

Das Präsentationsmodul besteht aus den folgenden Dateien:

- `prosetsif.bet` beinhaltet die Realisierung des erweiterten Systemmenüs.
- `prosetspec.bet` enthält die ProSet-spezifischen Kontextmenüs und die Aufrufe der entsprechenden in anderen Modulen realisierten Funktionalität.
- `prosetdialogs.bet` enthält die Realisierung der Dialogfenster *Symbol table*, *Semantic errors* und *Compiler options*.

Für die Realisierung des Präsentationsmoduls wurden Ausgaben in englischer Sprache verwendet, da die Ausgaben des BETA-Rahmenwerks ebenfalls nur in englischer Sprache erfolgen. Auf eine anpaßbare Gestaltung der Oberflächentexte (beispielsweise durch Nutzung einer externen Textdatei) wurde im Rahmen dieser Arbeit verzichtet. Hier muß eine endgültige Version des Rahmenwerks abgewartet werden.

Die Realisierung der Kontextmenüs erfolgt durch geschachtelte Muster. Die Menüeinträge bestehen im wesentlichen aus zwei anzupassenden Mustern: Ein Statusmuster bestimmt, ob der Menüeintrag aktivierbar oder inaktiv ist und ein Ausführungsmuster implementiert das auszuführende Verhalten. Der Programmausschnitt in Abbildung 17 enthält die Definition des Kontextmenüs für die Knoten

```
IdMenu: @mysimplemenu
 (#
   iNextAppl: @item
    (# Node: ^Pst.Id; A: ^AstInterface.Ast;
       DoGet:
         (# do cs.node.father->Node[]; Node.GetNextAppl->A[] #);
       onStatus:: (# do DoGet; (A[]->ValidNode)->value #);
       onSelect:: (# do A[]->ChangeFocus; #) #);
   iDecl: @item
    (# Node: ^Pst.Id; A: ^AstInterface.Ast;
       DoGet: (# Decl: ^Pst.IdDecl;
             do cs.node.father->Node[]; Node.GetDecl->A[];
                (if (A[]->Node.IsNodeExpanded)
                    and (A[]->Decl[]).InImplicitList then
                    Decl.GetFirstAppl->A[]; if) #);
       onStatus:: (# do DoGet; (A[]->ValidNode)->value #);
       onSelect:: (# do
                    (if A.Index<>Node.Index then
                         A[]->ChangeFocus;
                       else
                        (NONE, 'This is an\nimplicit declaration.', 'Notice:')
                            ->alertUser; if) #) #);
   open::  (# do
            'Goto next application'->iNextAppl.new;
            'Goto declaration'->iDecl.new; #) #);
```

Abbildung 18 Beispiel für die Realisierung eines Kontextmenüs

Id. Das Menü enthält zwei Einträge (vom Typ *item*), die durch die Muster *iDecl* und *iNextAppl* definiert werden. Jeder der beiden Einträge enthält das virtuelle Muster *onStatus* zur Definition der Aktivierbarkeit und das virtuelle Muster *onSelect* für den Aufruf der betreffenden Funktionalität.

5.2 Test der Module

Im folgenden wird die Vorgehensweise beim Testen der Module kurz beschrieben.

Test der Spezifikation für AST und Parser

Die Spezifikation der ASTs und des Parsers wurde anhand von Beispielprogrammen (u.a. aus der Sprachbeschreibung [DFG+92a]) getestet. Es wurde versucht, mit den Testprogrammen alle Sprachmittel abzudecken. Weitere Testläufe wurden später innerhalb der PU interaktiv durchgeführt.

Bei den Tests wurde ein Fehler innerhalb des MPS-Parsergenerators aufgedeckt, der zu fehlerhaft generierten Parsertabellen führte. Der Fehler innerhalb des Parsergenerators ist inzwischen durch Mjølner behoben worden.

Test der Unparsing-Spezifikation

Der Test der Unparsing-Schemata wurde anhand von Testprogrammen durchgeführt. Hier wurden insbesondere die Vollständigkeit des zu jedem Knoten definierten Schemas getestet. Bei komplexen Schemata, die Blockkonstrukte enthalten, wurde das Unparsing durch die Verwendung spezieller Testprogramme mit verschiedenen Fensterbreiten durchgeführt. Hierdurch konnte das Verhalten der Schemata für die verschiedenen Fälle des Unparsing-Algorithmus des MPS geprüft werden.

Test der Attributierung

Die Korrektheit der Attributierung wurde mit verschiedenen Testprogrammen überprüft. Dabei wurden der ermittelte Bereichsbaum, die Symbol- und Kreuzreferenztabellen in lesbarer Form angezeigt und analysiert.

Test der semantischen Verweise

Auch hier wurde die Ausgabe der Verweise (*definition-use-chain*) in lesbarer Form vorgenommen sowie die Konsistenz (*use-definition-chain*) überprüft.

Test der Transformationen

Die Transformationen wurden mit einem Testprogramm für verschiedene Testdaten durchgeführt. Das Ergebnis wurde mittels erneutem Unparsing und anhand der Ausgabe der Baumstruktur überprüft.

Test des Präsentationsmoduls

Die Menüs und Dialoge der Oberfläche wurden interaktiv getestet.

6 Zusammenfassung

Im folgenden werden die Ergebnisse der Diplomarbeit kurz zusammengefaßt. Es werden einerseits die Vor- und Nachteile des BETA-Metaprogrammiersystems zusammengetragen, andererseits wird die Eignung der Prototyping-Umgebung für die Verarbeitung von PROSET-Programmen beurteilt.

Beurteilung des BETA-Metaprogrammiersystems

Während der Verwendung des BETA-Metaprogrammiersystems sind einige vorteilhafte und nachteilige Eigenschaften dieses Systems offenbar geworden. Diese sind einerseits konzeptionell bedingt, andererseits durch die Ausrichtung auf BETA begründet.

Ein wesentlicher Kritikpunkt ist die gemeinsame Spezifikation von abstrakten Syntaxbäumen und dem Parser für Sprachen, die mit dem MPS Verwendung finden sollen. Die gemeinsame Spezifikation erfordert anfangs zwar einen geringeren Aufwand, allerdings führen die Einflüsse, die durch eine parserkonforme Spezifikation auf die Spezifikation von ASTs genommen werden, zu zusätzlichem Folgeaufwand.

Kritisch anzumerken ist ebenfalls die in mehreren Bereichen des MPS vorhandene rudimentäre Unterstützung. So sind im Bereich der lexikalischen Analyse sehr starke Einschränkungen gegeben und im Bereich des Unparsens sind für PROSET zusätzliche Konzepte wünschenswert. Im Bereich der Attributierung kann durch die ausschließlich angebotene operationale Spezifikation zwar die Flexibilität von BETA voll ausgenutzt werden, die Realisierung ist aber detailliert auszuformulieren und entsprechend fehleranfällig. Bei der Analyse von ASTs und der Bereitstellung von Transformationen macht sich die rein operationale Unterstützung weitaus stärker negativ bemerkbar. Routineaufgaben, wie die Prüfung einer Baumstruktur (*matching*), der Zugriff auf tief verschachtelte Baumstrukturen sowie die Spezifikation von Ersetzungen, sind durch die notwendige detaillierte Ausformulierung, bei der viele Überprüfungen durchgeführt werden müssen, schwer lesbar und ebenfalls fehleranfällig. In diesem Bereich ist deklarative Unterstützung unbedingt wünschenswert.

Zusammenfassend kann aus diesen Nachteilen für das MPS eine bedingte Eignung zur Erstellung von Entwicklungsumgebungen festgestellt werden.

Eignung der Prototyping-Umgebung

Für den sinnvollen praktischen Einsatz der PU sind einige zusätzliche Erweiterungen nötig, die im Rahmen dieser Diplomarbeit nicht realisiert werden konnten. Diese Erweiterungen sind im Bereich des Editors, der Datenintegration, der Kommunikation mit externen Werkzeugen und der Hilfefunktion vorzunehmen.

Im Bereich des Editors sind insbesondere die Unterstützung durch weitere Transformationen zu nennen. So kann beispielsweise die Editierung von Ausdrücken besser unterstützt werden, indem die vom MPS bereitgestellten Expandierungen durch zusätzliche Transformationen ergänzt werden, so daß Expandierungen über mehrere Stufen ermöglicht werden.

Ebenso ist die Erweiterung der Benutzerführung wünschenswert. So könnten die Menüs für Expandierungen schon gewisse weitergehende Kontexteinschränkungen reflektieren, z.B. die Auswahl der Anweisungen `quit` und `continue` ausschließlich innerhalb von Schleifen anbieten.

Auch bei der Editierung von Prozeduraufrufen könnte gewisse Hilfestellung bereits bei der Erstellung gewährt werden und so zu der Vermeidung von Fehlern führen. So ist die Erzeugung eines Aufrufmusters mit der richtigen Anzahl an Parametern und die Unterstützung bei der Einfügung von l-Werten für `wr`- und `rw`-Parameter denkbar.

Im Bereich der Datenintegration konnte die Mehrbenutzerfähigkeit und die Bereitstellung einer Versionsverwaltung wegen der Verwendung der hauptspeicherbasierten Repräsentation für abstrakte Syntaxbäume des MPS nicht erreicht werden. In diesem Bereich ist die Einbindung eines NSDBS, wie der objektorientierten Datenbank des BETA-Systems, eine sinnvolle Erweiterung. Die Integration eines NSDBS mit dem MPS muß allerdings in enger Zusammenarbeit mit Mjølner-Informatics erfolgen.

Im Bereich der Kommunikation mit externen Werkzeugen sind insbesondere die Nutzung der Dienste der PU durch andere Werkzeuge als Erweiterung zu nennen. Für die Realisierung dieser Eigenschaft ist in erster Linie eine Stabilisierung des BETA-Rahmenwerks, sowie die Offenlegung und zusätzliche Erweiterungsmöglichkeiten der Ablaufsteuerung des Rahmenwerks notwendig.

Im Bereich der Hilfefunktion ist neben der Bereitstellung von allgemeiner Hilfe zur Bedienung der Umgebung und Hilfe zu den sprachspezifischen Erweiterungen der Umgebung auch die Bereitstellung von Hilfe zu der Sprache ProSet selbst zu beachten. Auch in diesem Bereich sollte eine Stabilisierung des BETA-Rahmenwerks abgewartet werden.

A Installation und Benutzerhandbuch

Dieser Anhang beschreibt die Installation und die Arbeit mit der Prototyping-Umgebung in Form eines Benutzerhandbuches für die PROSET-spezifischen Funktionen der PU.

A.1 Installation

Die Verwendung der PU ist auf Sun Sparc Rechnern mit dem Betriebssystem Solaris 1 unter X11 möglich. Vor der Verwendung der PU müssen einige Vorbereitungen getroffen werden. Im folgenden werden dazu die benötigten Dateien für die PU selbst, sowie weitere benötigte Software-Systeme aufgeführt.

A.1.1 Benötigte Dateien der PU

Die PU besteht aus einer Reihe von Dateien, die teilweise in bestimmten Verzeichnissen vorhanden sein müssen. Die ausführbare Datei `prosetsif` und der Importfilter `import` können in einem beliebigen Verzeichnis installiert werden. Für die Verwendung der PU mit der Sprache PROSET müssen die Dateien der Grammatikbeschreibung durch die PU auffindbar sein. Dies wird erreicht durch die Installation der Konfigurationsdatei `MBSgrammars.text` in das Hauptverzeichnis des Benutzers (*home directory*) ~. Diese Datei muß einen Verweis auf die zu verwendenden Sprachbeschreibungsdateien enthalten. Abbildung 19 zeigt ein Beispiel für diese Datei. Sie enthält einen Verweis auf die weiter unten aufgeführten Grammatikdateien zu PROSET innerhalb des Verzeichnisses `~/PU`.

Folgende Dateien bilden die Grammatikbeschreibung der Sprache PROSET:

> `proset-meta.ast` enthält einen abstrakten Syntaxbaum der Sprachspezifikation.
> `proset-parser.btab` enthält die Parsertabelle.
> `proset-pretty.ptbl` enthält die Unparsingregeln in Form einer Tabelle.
> `proset.ast` ist eine intern durch das MPS verwendete Datei.

Für die Verwendung des Importfilters ist die Installation der Datei `import.awk` in das Verzeichnis der Grammatikdateien erforderlich.

```
[[
-- INCLUDE '~/PU/proset'
]]
```

Abbildung 19 Die Datei MBSgrammars.text

Zusätzlich ist die Definition der Umgebungsvariablen PROSETPU erforderlich. Diese Umgebungsvariable muß ebenfalls das Verzeichnis der Grammatikdateien bezeichnen.

A.1.2 Weitere benötigte Software-Systeme

Für die Verwendung der Persistenzwerkzeuge und des PROSET-Übersetzers wird der Zugriff auf weitere Software-Systeme benötigt.

Zur Verwendung der Persistenzwerkzeuge ist der Zugriff auf H-PCTE (Version 2.7) und das Vorhandensein einer H-PCTE-Datenbank zur Speicherung der *p-files* erforderlich. Zusätzlich muß der Zugriff auf die dynamischen Linkbibliotheken von *ToolTalk* gewährleistet sein, die mit *Openwindows* Version 3.0 verfügbar sind.

Der PROSET-Übersetzer muß in der Version 0.6 verfügbar sein.

Die Oberfläche der PU selbst benötigt den Zugriff auf Linkbibliotheken zu *Motif* Version 1.2.1 und den Zugriff auf das BETA-System der Version 3.1.

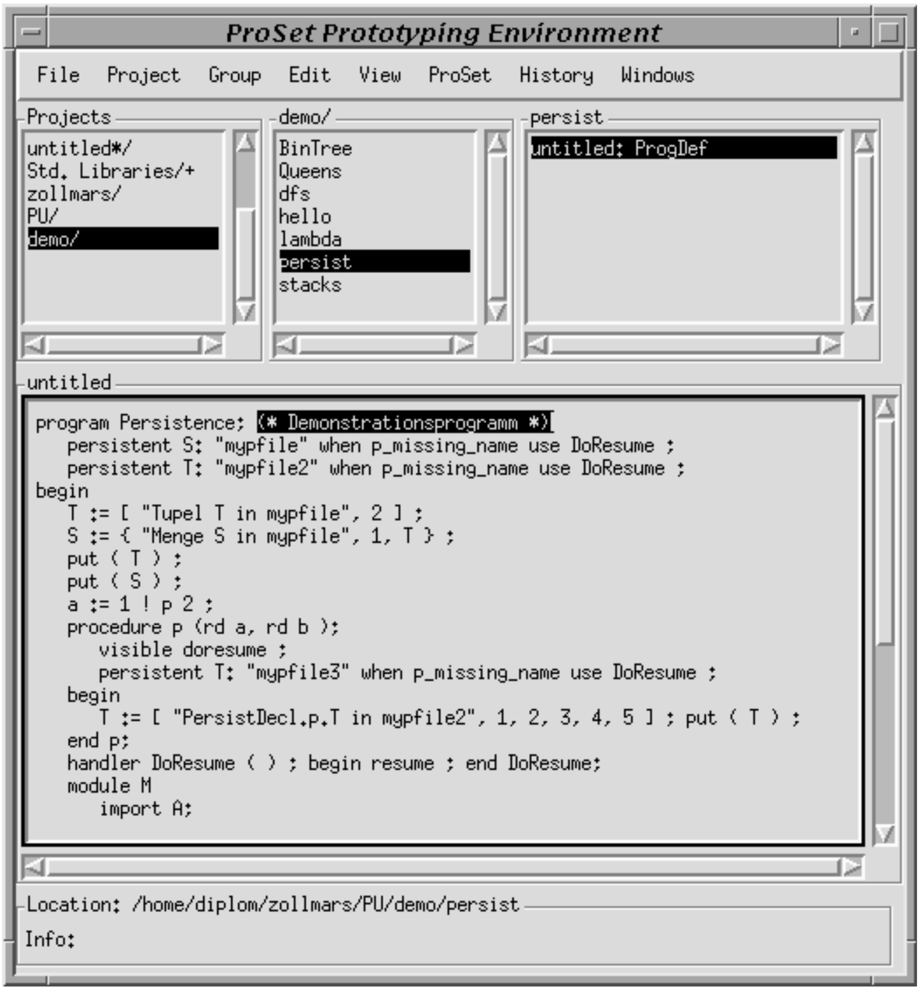


Abbildung 20 Bildschirmabzug: Die Prototyping-Umgebung

A.2 Benutzerhandbuch

Das Benutzerhandbuch soll die PROSET-spezifische Bedienung der PU erläutern. Die grundlegende Bedienung, die bereits durch das BETA-Rahmenwerk vorgegeben ist, wird hier vorausgesetzt und kann in dem Benutzerhandbuch zu Sif [Mjøl94a] nachgelesen werden.

Im folgenden werden die Bezeichnungen von Einträgen in Kontextmenüs in einer anderen `Schriftart` und sonstige Ausgaben der Oberfläche in *kursiver Schrift* gedruckt.

Abbildung 20 zeigt einen Bildschirmabzug der PU nach dem Öffnen eines PROSET-Programmes. Am oberen Rand des Fensters befindet sich das Systemmenü. Es enthält unter anderem den Eintrag `ProSet`, der PROSET-spezifische Menüeinträge zusammenfaßt. Dieses Menü wird im folgenden als PROSET-Systemmenü bezeichnet. Im oberen Bereich der PU werden drei Fenster zur Auflistung von Verzeichnissen des Dateisystems, zur Auflistung von Dateien und ein weiteres BETA-spezifisches Fenster angezeigt (von links nach rechts). Der untere Bereich dient der Editierung von PROSET-Programmen. Hier wird der Quelltext des Programmes *persist* aus dem Verzeichnis *demo* durch den Unparser dargestellt. Innerhalb des Programmes ist ein Kommentar selektiert worden.

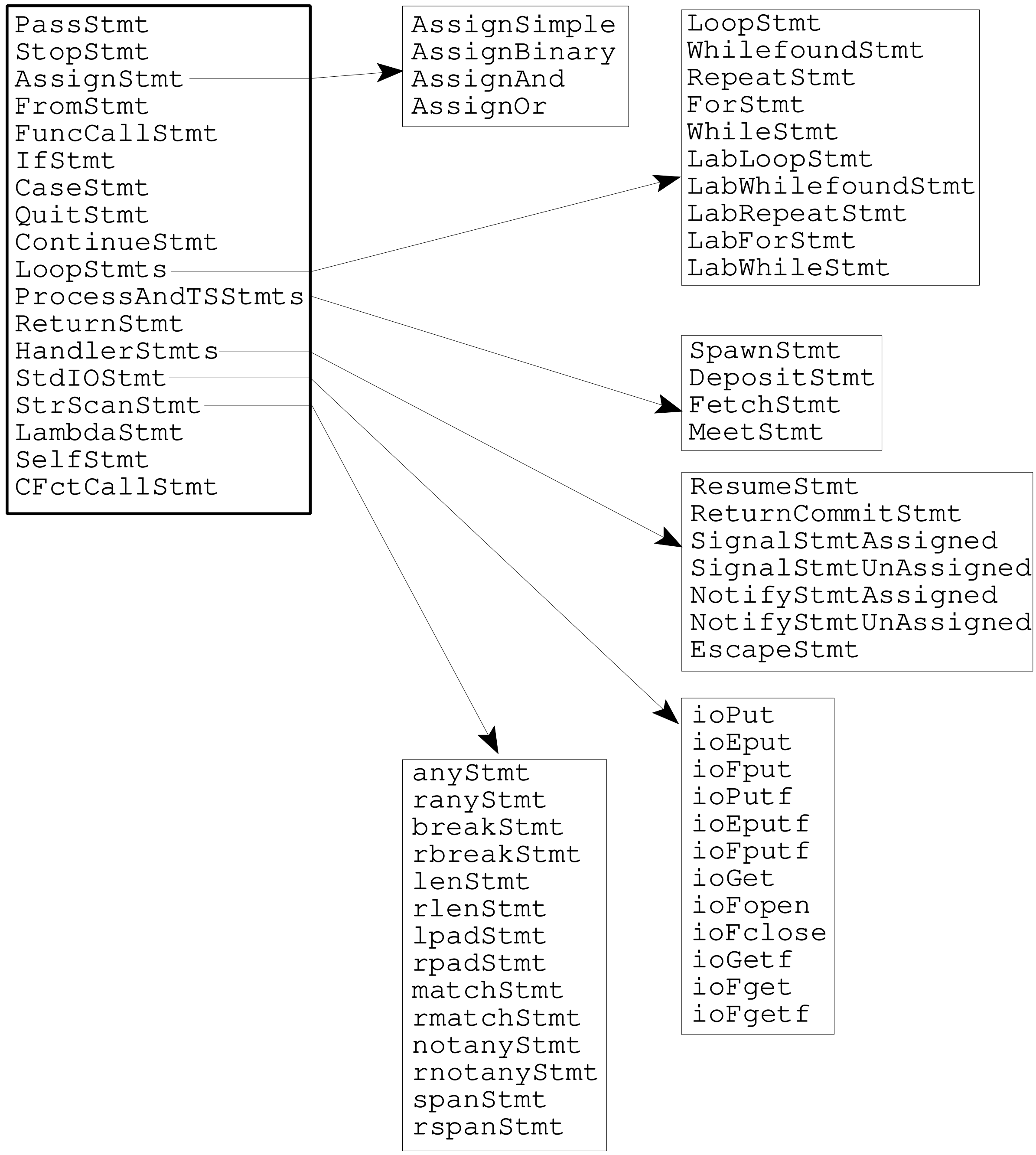


Abbildung 21 Kontextmenüs: Expandierung von Anweisungen

A.2.1 Editieren von PROSET-Programmen

Der Editor der PU arbeitet syntaxbasiert, neue Sprachkonstrukte werden durch sukzessive Ersetzung von Platzhaltern eingefügt. Platzhalter sind Inkrementnamen, die durch doppelte spitze Klammern eingefaßt sind (z.B. <<*Stmt*>> als Platzhalter für eine Anweisung). Wird innerhalb des Editors ein solcher Platzhalter selektiert, so kann für dieses Inkrement durch Drücken der rechten Maustaste ein Kontextmenü geöffnet werden. Dieser Abschnitt beschreibt die wichtigsten PROSET-spezifischen Kontextmenüs, die bei der Erstellung von Programmen verwendet werden.

Im folgenden werden die Kontextmenüs für die Expandierung von Anweisungen und die Kontextmenüs für die Expandierung von Ausdrücken beschrieben. In den zugehörigen Abbildungen sind die Kontextmenüs als Rechtecke dargestellt, die textuelle Einträge enthalten. Jede Zeile beschreibt dabei einen Menüeintrag. Verbindungspfeile zwischen einem Menüeintrag und einem Kontextmenü beschreiben eine mehrstufige Auswahl, d.h. bei Anwahl dieses Eintrages wird der selektierte Platzhalter durch den für das angewählte Inkrement ersetzt. Alle anderen Einträge fügen an Stelle des Platzhalters das entsprechende Sprachkonstrukt ein.

Expandierung von Anweisungen

Abbildung 21 zeigt die Kontextmenüs für die Expandierung von Anweisungen. Ausgehend von dem hervorgehobenen Kontextmenü für den Platzhalter *<<Stmt>>* in der oberen linken Ecke können Anweisungen wie z.B. `pass`, `stop` und `if` direkt durch Selektion der Einträge `PassStmt`, `StopStmt` und `IfStmt` ausgewählt werden. Zuweisungen, Schleifenkonstrukte, tupelraumbezogene Anweisungen, Anweisungen für Behandlungsroutinen, Ein-/Ausgabeanweisungen sowie Anweisungen für Verarbeitung von Zeichenketten sind in weiteren Kontextmenüs unter den Platzhalternamen `AssignStmt`, `LoopStmts`, `ProcessAndTSStmts`, `HandlerStmts`, `StdIOStmts` und `StrScanStmts` zusammengefaßt.

Expandierung von Ausdrücken

Abbildung 22 stellt die Kontextmenüs im Zusammenhang mit der Expandierung von Ausdrücken zusammen. In der oberen linken Ecke ist das Kontextmenü für den Platzhalter *<<Expr>>* dargestellt. Wie bereits beschrieben, ist die syntaxbasierte Editierung von arithmetischen Ausdrücken durch die hierarchische Spezifizierung der einzelnen Prioritätsstufen recht umständlich. Um beispielsweise einen Additionsausdruck zu erzeugen, müssen die Stufen geringerer Priorität durchlaufen werden und sukzessive die Menüeinträge `OrTerm`, `AndTerm`, `BoolTerm`, `SetTerm` und schließlich `AddExpr` der jeweiligen Kontextmenüs ausgewählt werden. Neben der hierarchischen Teilgrammatik für arithmetische Ausdrücke sind drei weitere Menüs in der Abbildung enthalten. Diese beschreiben Konstrukte auf die keine Selektion angewendet werden kann (ausgehend von `Primary`), Konstrukte auf die eine Selektion angewendet werden kann (Kontextmenü von `PrimarySelection`) und ein Kontextmenü für einfache Konstanten und Typbezeichner (Kontextmenü von `SimpleKeywords`).

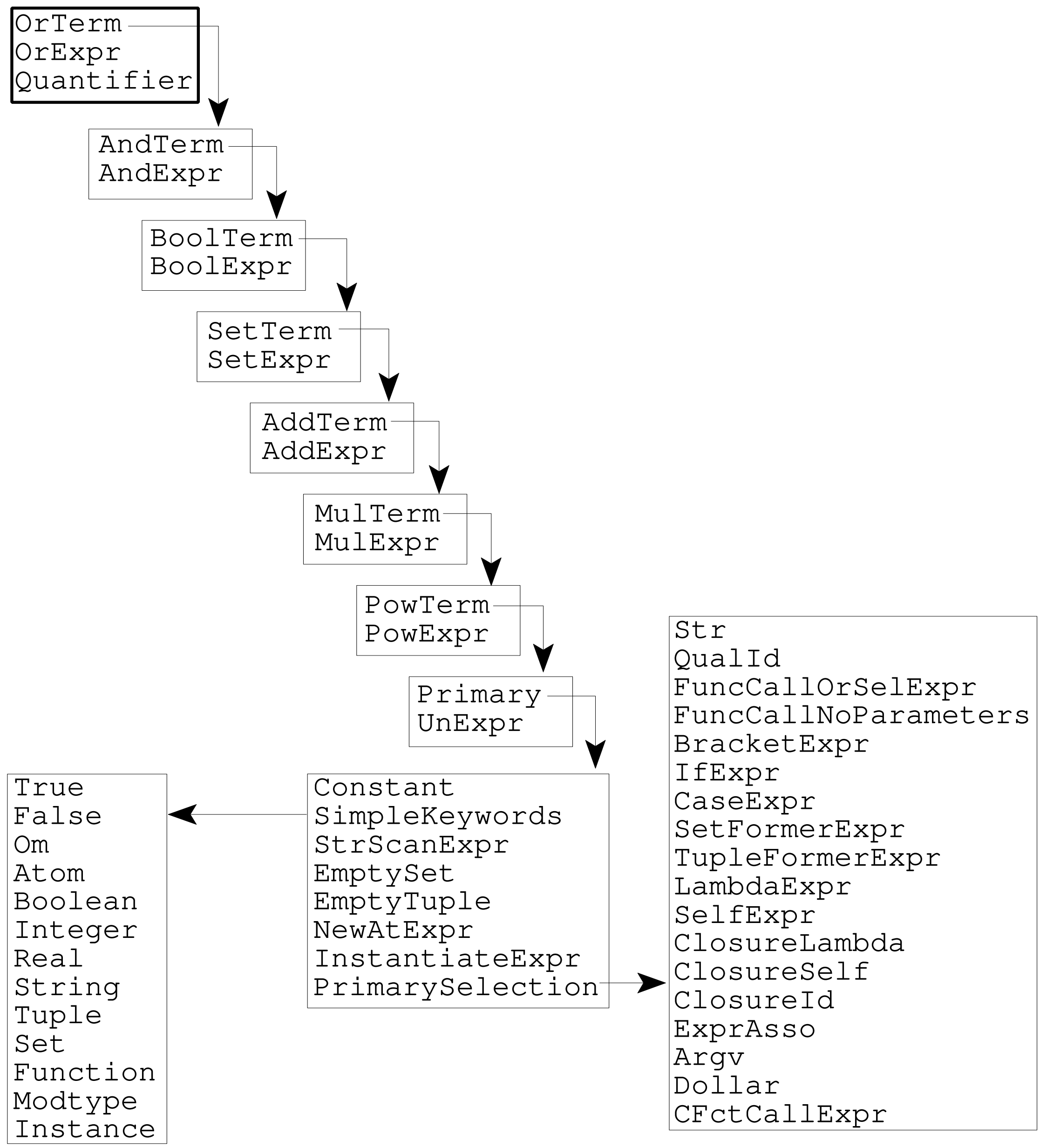


Abbildung 22 Kontextmenüs: Expandierung von Ausdrücken

A.2.2 Statische Analyse

In den folgenden Abschnitten wird die Durchführung der statischen Analyse und die Nutzung der semantischen Informationen in der PU beschrieben.

Die statische Analyse des bearbeiteten Programmes wird durch Anwahl des PROSET-Systemmenüeintrages `Check source` aufgerufen. Nach der Durchführung der Analyse stehen die Analyseergebnisse für den Benutzer zur Verfügung. Das Fenster *Semantic errors* wird automatisch angezeigt, falls Fehler festgestellt wurden. Falls keine Fehler festgestellt wurden, wird dies dem Benutzer durch ein Benachrichtigungsfenster angezeigt.

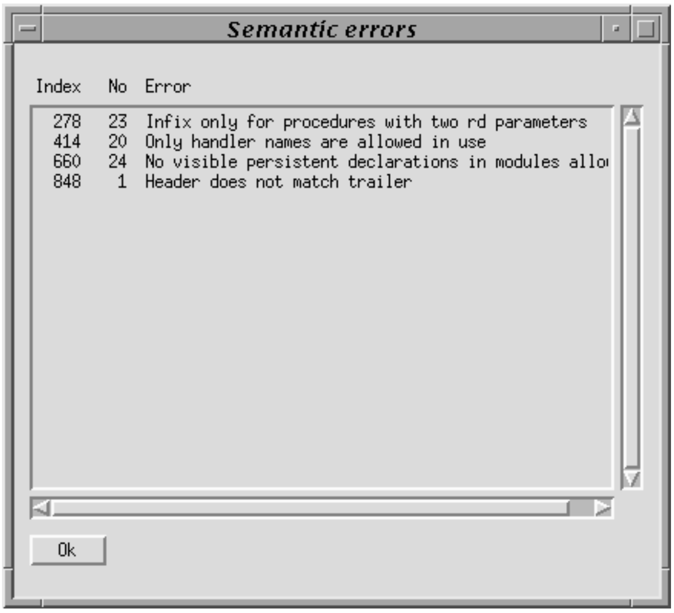


Abbildung 23 Bildschirmabzug: Das Fenster *Semantic Errors*

An weiteren Analyseergebnissen kann der Benutzer die Symboltabelle für Bereiche in Form des Fensters *Symbol table* und die Informationen der Kreuzreferenztabelle als Querverweise an Namensanwendungen und Namensdeklarationen in Form von Kontextmenüs nutzen.

A.2.2.1 Das Fenster *Semantic errors*

Das Fenster *Semantic errors* beinhaltet eine Liste der bei der statischen Analyse erkannten Fehler. Jeder Listeneintrag zeigt die interne Referenz des fehlerhaften Knotens, die Fehlernummer und die textuelle Fehlermeldung. Durch Anwahl eines der Einträge wird innerhalb der PU der fehlerhafte Knoten in den Darstellungsbereich gebracht und markiert. Falls der Knoten zwischenzeitlich durch Änderungen des AST gelöscht wurde, so wird dies durch ein Benachrichtigungsfenster angezeigt.

Abbildung 23 zeigt den Bildschirmabzug eines Fehlerfensters. In der linken, mit *Index* bezeichneten Spalte der Fehlerliste werden die Knotennummern der fehlerhaften Knoten angezeigt, die folgende, mit *No* bezeichnete Spalte enthält die Fehlernummern auf die in der rechten, mit *Error* bezeichneten Spalte folgend die Fehlermeldungen als Text angezeigt werden.

A.2.2.2 Das Fenster *Symbol table*

Das Fenster *Symbol table* wird durch Anwahl des ProSet-Systemmenüeintrages `Symbol table` angezeigt. Es listet die deklarierten Symbole des Bereiches auf, innerhalb dessen sich die Markierung bei Aufruf befindet. Die neue Markierung zeigt den Bereich an, dessen Symboltabelle dargestellt wird.

Die Einträge der Liste sind nach global deklarierten, lokal deklarierten, implizit verwendeten und schließlich nicht-lokal verwendeten Bezeichnern gruppiert. Diese Gruppierung wird durch die Attribute *visible*, *hidden*, *implicit* und *inherited* kenntlich gemacht. Dieses Deklarationsattribut folgt auf den Namen des Bezeichners. Abschließend werden weitere Deklarationseigenschaften des Bezeichners in einer geklammerten Liste angezeigt. Diese kann die Attribute *procedure*, *handler* oder *module* für die Deklaration von Prozeduren, Modulen, Behandlungsroutinen,

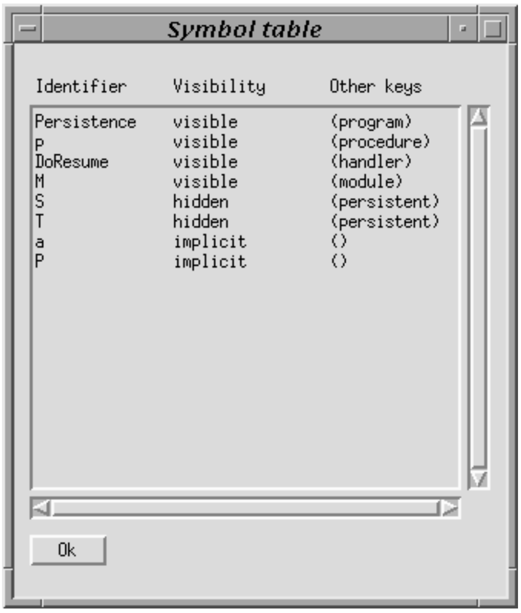


Abbildung 24 Bildschirmabzug: Das Fenster *Symbol table*

beziehungsweise *persistent* und *constant* für persistent oder konstant deklarierte Bezeichner enthalten. Abbildung 24 zeigt den Bildschirmabzug eines Symboltabellenfensters. Die Namen der Spalten sind mit *Identifier*, *Visibility* und *Other keys* bezeichnet.

A.2.2.3 Die Nutzung der semantischen Attribute

Semantische Attribute werden in Form von Querverweisen über Kontextmenüs angeboten. Wird durch die aktuelle Markierung eine Namensdeklaration bezeichnet, so kann durch die Anwahl des Eintrages `Goto first application` des Kontextmenüs die erste Anwendung dieses Namens erreicht werden. Diese Namensanwendung wird dann markiert. Von markierten Namensanwendungen aus kann einerseits durch Anwahl des Kontextmenüeintrages `Goto next application` die nächste Anwendung dieses Namens erreicht werden. Die Reihenfolge ist durch die Position innerhalb des AST vorgegeben. Andererseits kann durch Anwahl von `Goto declaration` der Deklarationsknoten erreicht werden. Ist der Name implizit deklariert, so wird die erste Verwendung dieses Namens markiert. Falls der Menüeintrag von der ersten Verwendung aus aufgerufen wird, so erscheint ein Benachrichtigungsfenster, das dem Benutzer mitteilt, daß es sich um eine implizite Deklaration handelt.

Falls keine Analyseergebnisse zur Verfügung stehen oder eine Deklaration nicht verwendet wird, so sind die Kontextmenüeinträge inaktiv und können nicht angewählt werden.

A.2.3 Transformationen

Transformationen werden für die Umwandlung von Schleifen und bedingten Anweisungen angeboten. Die folgenden Abschnitte beschreiben die Verwendung dieser Transformationen.

A.2.3.1 Transformation von Schleifen

Die Transformationen für die Umwandlung von Schleifen mit oder ohne Sprungmarken setzen die Markierung eines der folgenden Konstrukte voraus: `repeat`-Anweisung, `while`-Anweisung, `loop`-Anweisung, `for`-Anweisung oder `whilefound`-Anweisung. Da die Schleifenkonstrukte über geschachtelte Inkremente modelliert wurden, muß die Markierung neben dem Schleifenbereich auch das abschließende `end` umfassen.

An diesen Konstrukten stehen Transformationen für die Ergänzung beziehungsweise das Löschen der Sprungmarken durch Anwahl des Kontextmenüeintrages `Replace with labeled statement` oder `Replace with unlabeled statement` zur Verfügung.

Weitere Transformationen bieten die Kontextmenüs der `repeat`-Anweisungen und `while`-Anweisungen an. Diese Schleifenanweisungen können durch die Auswahl der Menüeinträge `Replace with while` bzw. `Replace with repeat` in ihr Gegenstück transformiert werden. Hierbei handelt es sich nicht um semantische Transformationen.

A.2.3.2 Transformation bedingter Anweisungen und Ausdrücke

Für die Umwandlung von `if`-Konstrukten in `case`-Konstrukte und umgekehrt stehen in den Kontextmenüs Einträge für folgende semantische Transformationen zur Verfügung:

- an `if`-Anweisungen der Menüeintrag `Replace with case statement`
- an `if`-Ausdrücken der Menüeintrag `Replace with case expression`
- an `case`-Anweisungen der Menüeintrag `Replace with if statement`
- an `case`-Ausdrücken der Menüeintrag `Replace with if expression`

Eine weitere Transformation steht für den Tausch von Anweisungslisten zwischen den einzelnen Fällen der `if`-Anweisungen zur Verfügung. Sie wird aufgerufen durch die Auswahl des Kontextmenüeintrages `Swap statements with next alternative`:

- An markierten `if`-Anweisungen besteht die Möglichkeit die Anweisungsliste nach `then` mit derjenigen des nächsten `elseif`-Falles oder des `else`-Falles zu tauschen.
- An markierten `elseif`-Fällen kann ebenfalls die Anweisungsliste nach `then` mit derjenigen des nächsten `elseif`-Falles oder des `else`-Falles getauscht werden. Wird die Transformation für den letzten `elseif`-Fall angewählt und existiert kein `else`-Fall, so wird die Anweisungsliste mit derjenigen der `if`-Anweisung getauscht.
- An markierten `else`-Fällen kann die Anweisungsliste mit derjenigen der `if`-Anweisung nach `then` getauscht werden.

A.2.4 Nutzung der Persistenzwerkzeuge

Der Aufruf der Persistenzwerkzeuge wird durch die Kontextmenüs von persistenten Deklarationen ermöglicht. Als Vorbedingung muß die Bezeichnung des *p-files* in der Deklaration mittels einer literalen Zeichenkette erfolgen. Dort kann durch Auswahl des Menüeintrages `View/modify persistent value` der *p-file editor* und durch Auswahl des Eintrages `Create p-file` das *p-file tool* aufgerufen werden.

A.2.5 Aufruf des Übersetzers

Durch Aufruf des ProSet-Systemmenüeintrages `Compile source` wird der ProSet-Übersetzer für das aktuell bearbeitete Programm gestartet. Die Übersetzung ist nur dann möglich, wenn das aktuelle Programm vollständig ist und zuvor das Programm geprüft wurde und keine Fehler enthält.

Über den ProSet-Systemmenüeintrag `Compiler options` können Optionen definiert werden, die bei der Übersetzung in das Kommando für den Aufruf des Übersetzers eingebunden werden.

A.2.6 Importierung externer Quelldateien

Zur Benutzung mit der PU müssen die Quelldateien in ein von der PU unterstütztes Format konvertiert werden. Durch die Verwendung des Filters zur Importierung werden `@include`-Direktiven und `macro`-Verwendungen aus der Quelldatei eliminiert, sowie Kommentare mit `--` umgewandelt in Kommentare mit `(*` und `*)`.

Die Importierung erfolgt entweder von der Kommandozeile oder innerhalb der PU. Der Aufruf von der Kommandozeile erfolgt durch Angabe des Kommandos `import`. Die zu importierende Datei muß als erster Parameter an den Filter übergeben werden. Der zweite Parameter bezeichnet die Zieldatei für die bearbeiteten Ergebnisse.

Innerhalb der PU wird die Importierung durch Aufruf des Menüpunktes `Import` angestoßen. Der Name des zu importierenden Programmes wird innerhalb eines Dateiauswahldialogs festgelegt. Der importierte Programmtext wird als Datei *untitled.pst* erzeugt und kann nach dem Öffnen unter einem beliebigen Namen gespeichert werden.

Literaturverzeichnis

[Balz93] H. Balzert: *CASE - Systeme und Werkzeuge*, BI-Wissenschaftsverlag 1993

[BK91] N.S. Barghouti, G.E. Kaiser: *Concurrency Control in Advanced Database Applications*, ACM Computing Surveys 1991

[BKKZ92] R. Budde, K. Kautz, K. Kuhlenkamp, H. Züllighoven: *Prototyping: An Approach to Evolutionary System Development*, Springer 1992

[BP92] W.R. Bischhofberger, G. Pomberger: *Prototyping-oriented Software Development - Concepts and Tools*, Springer 1992

[Bubo96] A. Bubolz: *Generierung eines syntaxgesteuerten Editors für ProSet mit dem Synthesizer-Generator*, Diplomarbeit, Universität Dortmund, voraussichtlicher Erscheinungstermin: 1996

[Boeh86] B.W. Boehm: *A Spiral Model of Software Development and Enhancement*, Software Engineering Notes, Vol. 11, 1986

[CI84] R.D. Cameron, M.R. Ito: *Grammar-Based Definition of Metaprogramming Systems*, ACM Transactions on Programming Languages and Systems, Vol. 6, No. 1, Januar 1984

[DF89] E.-E. Doberkat, D. Fox: *Software-Prototyping mit SETL*, Leitfäden und Monographien der Informatik, Teubner 1989

[DFG+92a] E.-E. Doberkat, W. Franke, U. Gutenbeil, W. Hasselbring, U. Lammers, C. Pahl: *ProSet - Prototyping with Sets, Language Definition*, Informatik-Bericht 02-92, Universität Essen, April 1992

[DFG+92b] E.-E. Doberkat, W. Franke, U. Gutenbeil, W. Hasselbring, U. Lammers, C. Pahl: *ProSet - A Language for Prototyping with Sets*, in Proceedings of the Third International Workshop on Rapid System Prototyping, S. 235-248, IEEE Computer Society Press, Juni 1992

[DKLM84] V. Donzeau-Gouge, G. Kahn, B. Lang, B. Melese: *Document Structure and Modularity in MENTOR*, SIGPLAN Notices, Vol. 19, No. 5, S. 141-148, 1984

[ES89] G. Engels, W. Schäfer: *Programmentwicklungsumgebungen, Konzepte und Realisierung*, Teubner 1989

[Floy84] C. Floyd: *A Systematic Look at Prototyping*, in: R. Budde et al.: Approaches to Prototyping, Springer 1984

[GH83] G. Goos, J. Hartmanis: *DIANA: An Intermediate Language for ADA*, Lecture Notes in Computer Science, No. 161, Springer 1983

[GHHM+92] M. Gera, B. Hirsch, B. Holtkamp, J.-P. Moularde, G. Samuel, H. Weber: *Eureka Software Factory - CoRe: A Conceptual Reference Model for Software Factories*, Eureka Software Factory 1992

[GJM91] C. Ghezzi, M. Jazayeri, D. Mandrioli: *Fundamentals of Software Engineering*, Prentice Hall 1991

[HN86] A.N. Habermann, D. Notkin: *Gandalf: Software Development Environments*, IEEE Transactions on Software Engineering 1986, S. 1117-1127

[HW87] R.C. Houghton, D.R. Wallace: *Characteristics and Functions of Software Engeneering Environments: An Overview*, Software Engineering Notes, Vol. 12, No. 1, Januar 1987

[Kapp95] C. Kappert: *Integration von Persistenzkonzepten in eine Prototyping-Sprache und Realisierung mit Hilfe eines Nicht-Standard-Datenbanksystems*, Diplomarbeit, Universität Dortmund 1995

[Kelt93] U. Kelter: *Integrationsrahmen für Software-Entwicklungsumgebungen*, Informatik-Spektrum 1993, 16: 281-285

[KLLM94] J. L. Knudsen, M. Löfgren, O. Lehrmann-Madsen, B. Magnusson: *Object Oriented Environments: The Mjølner Approach*, Prentice Hall 1994

[KLSZ92] A. Kieback, H. Lichter, M. Schneider-Hufschmidt, H. Züllighoven: *Prototyping in industriellen Software-Projekten*, Informatik-Spektrum 1992, 15: 65-77

[KSM94] J. L. Knudsen, E. Sandvad, S. Minör: *Grammar Based Architectures*, Introduction to Part VI in [KLLM94]

[McCa62] J. McCarthy: *LISP 1.5 Programmer's Manual*, The MIT Press, Cambridge, Mass. 1962

[Merl96] M. Merl: *Spezifikation und Visualisierung eines Style-Guides für graphische Benutzungsoberflächen der Werkzeuge einer Prototyping-Umgebung*, Diplomarbeit, Universität Dortmund 1996

[Mjøl94a] Mjølner Informatics: *Sif - A Hyper Structure Editor, Tutorial and Reference Manual*, Mjølner Informatics Report MIA90-11(1.2) 1994

[Mjøl94b] Mjølner Informatics: *The Mjølner BETA System: Metaprogramming System, Reference Manual*, Mjølner Informatics Report MIA91-14(1.2) 1994

[Mjøl94c] Mjølner Informatics: *The Mjølner BETA System: BETA Language Introduction*, Mjølner Informatics Report MIA94-26(1.0) 1994

[Mjøl95] Mjølner Informatics: *The Mjølner BETA System: Lidskjalv User Interface Framework, Reference Manual*, Mjølner Informatics Report MIA94-27(1.0) 1995

[MMPN93] O. L. Madsen, B. Møller-Pedersen, K. Nygaard: *Object-Oriented Programming in the BETA Programming Language*, Addison Wesley 1993

[Nagl93] M. Nagl: *Software-Entwicklungsumgebungen: Einordnung und zukünftige Entwicklungslinien*, Informatik-Spektrum 1993, 16: 273-280

[PG95] Projektgruppe 240: *Scotland Yard - Evaluation und Entwurf von Werkzeugen für eine Entwicklungsumgebung zum Prototyping*, Abschlußbericht der Projektgruppe, Universität Dortmund 1995

[Pohl96] H. Pohland: *Ein graphischer Debugger für PROSET-Linda*, Diplomarbeit, Universität Essen, Voraussichtlicher Erscheinungstermin: 1996

[RT87] T. Reps, T. Teitelbaum: *Language Processing in Program Editors*, Computer, Nov. 1987, S. 29-37

[Sun91] Sun Microsystems: *ToolTalk 1.1.1 User's Guide*, Sun Microsystems 1991-93

[TM81] W. Teitelman, L. Masinter: *The InterLisp Programming Environment*, Computer, April 1981

[TN92] I. Thomas, B. A. Nejmeh: *Definitions of Tool Integration for Environments*, IEEE Software, März 1992, S. 29-35

[WG84] W.M. Waite, G. Goos: *Compiler Construction*, Springer 1984

[WJ93] L. Wakeman, J. Jowett: *PCTE - The Standard for Open Repositories*, Prentice Hall 1993

[WM92] R. Wilhelm, D. Maurer: *Übersetzerbau - Theorie, Konstruktion, Generierung*, Springer 1992