Spezifikation und Integration von Vor- und Nachbedingungen für Aufgaben in CoMo-Kit

Andreas Raquet
Im Krummen Rain 24
67661 Kaiserslautern
raquet@informatik.uni-kl.de

27. Oktober 1997

Inhaltsverzeichnis

1 Einleitung und Motivation

1.1 Problematik von Entwicklungsprozessen

Viele Entwicklungsprozesse, wie sie z.B. beim Entwurf von großen Softwaresystemen benötigt werden, basieren in erster Linie auf dem Wissen der mit der Entwicklung betrauten Mitarbeiter. Mit wachsender Komplexität der Entwurfsaufgaben und mit wachsender Anzahl der Mitarbeiter in einem Projekt wird die Koordination und Verteilung dieses Wissens immer problematischer. Aus diesem Grund versucht man zunehmend, das Wissen der Mitarbeiter in elektronischer Form, d.h. in Rechnern zu speichern und zu verwalten. Dadurch, daß der Entwurf eines komplexen Systems ebenfalls am Rechner modelliert wird, steht benötigtes Wissen sofort zur Verfügung und kann zur Entscheidungsunterstützung herangezogen werden. Gerade bei der Planung großer Projekte stehen jedoch oft Entscheidungen aus, die erst später, während der Abwicklung getroffen werden können. Da gängige Workflow-Management-System zumeist eine komplette Modellierung verlangen, bevor die Abwicklung eines Projektmodells beginnen kann, habt sich dieser Ansatz gerade für umfangreiche Projekte als eher ungeeignet herausgestellt.

1.2 Das Projekt CoMo-Kit

In der Arbeitsgruppe "Künstliche Intelligenz" der Universität Kaiserslautern ist im Projekt **CoMo-Kit** ein entscheidungsunterstützendes System entstanden, das z.B. zur Planung von Softwareentwicklungsprozessen oder zum Entwurf von Bebauungsplänen eingesetzt werden kann. Der Entwicklungsprozeß wird zunächst soweit nötig modelliert und kann dann bereits teilweise abgewickelt werden, noch bevor die Modellierung beendet ist. Modellierung und Abwicklung gehen also verzahnt vonstatten.

1.3 Modellierung von Entwurfsprozessen in CoMo-Kit

Ein gesamter Entwurfsprozeß wird in **CoMo-Kit** zunächst als `Task` modelliert, also als zu erfüllende Aufgabe, die gewisse Eingaben erwartet und mindestens eine Ausgabe, das zu entwerfende *Produkt*, zurückliefert. Diese *Aufgabe* wird schrittweise verfeinert zu *Teilaufgaben*, die in sich geschlossene Teile des Entwurfs repräsentieren, also z.B. Teile die von genau einem Mitarbeiter erledigt werden. Dabei kann eine *Aufgabe* auch durch mehrere parallele *Teilaufgaben* verfeinert werden kann, wodurch alternative Möglichkeiten zum Lösen eines Problems modelliert werden können. Zwischen den *Teilaufgaben* besteht ein Datenfluß, der die Zwischen- bzw. Nebenprodukte darstellt, die während der späteren Abwicklung entstehen und von anderen *Teilaufgaben* benötigt werden. Folgendes Beispiel soll das Konzept anhand des Entwurfs eines modularen Compilers erläutern.

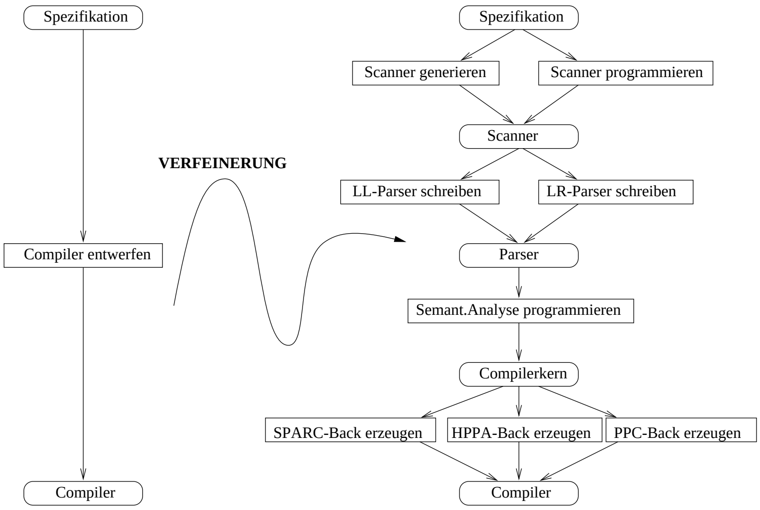

Abbildung 1: Modell eines Compilerentwurfs

Abb. 1 zeigt, wie die *Aufgabe* des Compilerentwurfs in verschiedene *Teilaufgaben* aufgespalten wird. Dabei wird bei jeder *Teilaufgabe* ein Modul des Compilers entworfen, also z.B. der Scanner oder der Parser. Jede dieser *Teilaufgaben* liefert als Ausgabe das durch sie entwickelte Modul, das von der nächsten *Teilaufgabe* als Eingabe verwendet wird. Die letzte *Teilaufgabe* liefert schließlich den gewünschten Compiler. Je nach Komplexität des Problems können beliebige *Teilaufgaben* beliebig weiter verfeinert werden.

1.4 Die Abwicklung von CoMo-Kit-Modellen

Die Abwicklung eines **CoMo-Kit**-Planes besteht im Wesentlichen in der Delegation der modellierten *Teilaufgaben* durch einen Projektleiter. Der Mitarbeiter, an den eine *Aufgabe* delegiert wurde, beginnt dann mit der Abarbeitung dieser *Aufgabe*. Dazu verwendet er z.B. Ergebnisse anderer Aufgaben, die bereits von anderen Mitarbeitern bearbeitet wurden, und die ihm in Form von Eingangsparametern für seine *Aufgabe* von **CoMo-Kit** zur Verfügung gestellt werden. Ist seine *Aufgabe* beendet, gibt er wiederum das Ergebnis seiner Arbeit durch die Belegung von Ausgangsparametern an **CoMo-Kit** weiter. Dadurch können dann evtl. weitere Aufgaben delegiert und abgewickelt werden.

Das System trägt dafür Sorge, daß nur *Teilaufgaben* delegiert werden können, die bereits vollständig modelliert sind. Zudem können *Teilaufgaben* nur dann ausgeführt werden, wenn alle von ihnen benötigten Eingabeparameter existieren. Warum dies zunächst sinnvoll erscheint soll wieder am Beispiel des modularen Compilers erläutert werden. Die Abarbeitung des vorgestellten Modells beginnt zunächst mit der Delegation der *Aufgabe* SScanner-Entwurfän einen Programmierer. Dieser Implementiert den Scanner und stellt ihn als Ausgabe zur Verfügung. Erst jetzt macht es Sinn als nächste *Teilaufgabe* den Parser zu implementieren, denn der ist stark auf Funktionsweise und Ausgabeformat des Scanners angewiesen. Die weitere Entwicklung des Compilers besteht in der sukzessiven Delegation und Abarbeitung der einzelnen *Teilaufgaben* bis der Programmierer des Backends den kompletten Compiler als Ausgabe liefert.

Anm.: In einem korrekten Entwurfsprozeß sollte zunächst das Interface zwischen Scanner und Parser spezifiziert werden, bevor mit der Implementierung des Scanners begonnen wird. Daher könnte die Implementierung des Parsers auch schon vor der Fertigstellung des Scanners beginnen. Eine Unterteilung in Spezifikation und Implementierung wäre jedoch für ein einfaches Beispiel zu aufwendig. Der Übersicht halber soll daher hier auf eine komplexere Modellierung verzichtet werden.

Die zunächst vernünftige Bedingung, daß Eingangsparameter einer *Teilaufgabe* vor der Abwicklung dieser *Aufgabe* existieren müssen erscheint bei näherer Betrachtung jedoch unzulänglich, wie folgende Betrachtungen anhand des bereits eingeführten Beispiels Compilerentwurfíllustrieren sollen.

1. Vor dem Entwurf des Compilerkerns, ist die Spezifikation eines Parsers vonnöten, dabei kann entweder ein LL-Parser oder ein LR-Parser zum Einsatz kommen. Im oben dargestellten Modell werden die Entwurfsaufgaben "LL-Parser erzeugen" und "LR-Parser erzeugen" zwar getrennt als Alternativen dargestellt, das Ergebnis jedoch nur in Form eines Parameters "Parser" modelliert. Es kann jedoch durchaus sinnvoll sein, LR- und LL-Parser als verschiedene Parameter zu modellieren, von denen nur einer notwendig ist für den Entwurf des Kerns. Bereits hier scheitert der **CoMo-Kit**-Ansatz, da es nicht möglich ist, eine Disjunktion von Eingabeparametern zu verlangen.

2. Da LR-Parser prinzipiell mächtiger sind als LL-Parser hängt die Art des verwendeten Parsers maßgeblich von der zu übersetzenden Grammatik ab. Für eine nicht LL-parsbare Grammatik ist ein LL-Parser unbrauchbar. Wiederum läßt sich diese Bedingung im **CoMo-Kit**-System nicht ohne gravierende Änderungen im Modell ausdrücken.

1.5 Das Projekt MILOS

Da der Entwurf großer Softwarepakete einer der Anwendungsschwerpunkte des **CoMo-Kit**-Systems ist, wurde im Rahmen des **MILOS**-Projektes eine Erweiterung des **CoMo-Kit**-Systems spezifiziert um besonderen Anforderungen beim Entwurf von Softwaresystemen gerecht zu werden. Eine vollständige Erläuterung findet sich in [1].

Unter anderem wird in **MILOS** ein Konzept spezifiziert, daß es dem Modellierer erlaubt, Vor- und Nachbedingungen für *Teilaufgaben* zu formulieren. Diese Bedingungen stellen Kontrollstrukturen dar, mit deren Hilfe es möglich wird, die Abwicklung des Modells zu steuern.

Im letzten Abschnitt hatten sich die Ausdrucksmöglichkeiten des **CoMo-Kit**-Systems bereits als unzulänglich erwiesen. Mit den erweiterten Möglichkeiten aus dem **MILOS**-Projekt läßt sich jedoch folgende Vorbedingung der *Aufgabe* "Kern implementieren" formulieren:

`'LR-Parser existiert ODER ( LL-Parser existiert und Grammatik ist LL-parsebar)'`

1.6 Diese Projektarbeit

Im **MILOS**-Projekt wurde für Vor- und Nachbedingungen ein sehr einfaches Konzept vorgestellt und spezifiziert. Die Aufgabe dieser Projektarbeit bestand darin, die bestehende Spezifikation zu überarbeiten und an das aktuelle **CoMo-Kit** System, insbesondere das aktuelle Typsystem, anzupassen. Kapitel 2 wird ausführlich die neue Spezifikation vorstellen und erläutern. Sie unterscheidet sich z.T. erheblich von der ursprünglichen **MILOS**-Spezifikation. Als nächstes war ein graphisches Interface zu implementieren, das es dem Benutzer erlaubt, Vor- und Nachbedingungen auf bequeme Art und Weise einzugeben. Kapitel 3 wird das Interface und seine Funktionsweise sowie die Anbindung an das Modell näher erläutern. Schließlich war ein Ausführungsmechanismus für Vor- und Nachbedingungen zu implementieren und in das Laufzeitsystem von **CoMo-Kit** zu integrieren. Kapitel 4 wird auf die Funktionsweise und die Integration in **CoMo-Kit** näher eingehen.

2 Spezifikation der Sprache für Vor- und Nachbedingungen

Die folgenden Überlegungen sowie die Spezifikation gelten für Vorbedingung und Nachbedingungen einer *Aufgabe*. Der Einfachheit halber soll in diesem Kapitel jedoch nur von Vorbedingungen die Rede sein.

2.1 Vorüberlegungen

Die Vorbedingung einer *Aufgabe* wird in Form eines Regulären Ausdrucks mit Bool'schem Rückgabewert, also `TRUE` oder `FALSE`, formuliert. In **MILOS** wurde für diese Regulären Ausdrücke eine einfache Spezifikation vorgestellt, ähnlich wie sie in Programmiersprachen wie C zu finden ist. Es existiert eine feste Menge von Basistypen, nämlich `NUMBER`, `BOOL`, `SYMBOL`, und `STRING`. Diese können durch eine vorgegebene Menge an arithmetischen und logischen Operatoren verknüpft werden. Zusätzlich stehen komplexe Produkte (ähnlich RECORDs bzw. structs in gängigen Sprachen) zur Verfügung. Diese können jedoch nur auf Gleichheit getestet werden.

Im Rahmen dieser Projektarbeit wurde die gesamte Spezifikation neu erarbeitet. Im folgenden sollen einige Überlegungen vorgestellt werden, die eine Neuerarbeitung notwendig gemacht haben.

- **erweiterbares Typsystem**
  Bereits in **MILOS** war ein erweiterbares Typsystem vorgesehen. Zur sinnvollen Verwendung erweiterter Typen sind jedoch Operatoren notwendig, die auf Instanzen dieser erweiterten Typen arbeiten können. Gängige Programmiersprachen stellen hierzu das Konzept der Operatorüberladung zur Verfügung. Ein solches Überladungskonzept wurde auch in die neue Spezifikation aufgenommen.

- **erweiterbares Operatorkonzept**
  Die **MILOS**-Spezifikation hat eine feste Menge von Grundoperatoren zur Verfügung gestellt. Die Erfahrung hat jedoch gezeigt, daß selbst eine umfassende Menge an Grundoperatoren an ihre Grenzen stoßen kann. Mit zunehmender Komplexität einer Vorbedingung werden komplexe Funktionen notwendig. Daher wurde ein Konzept erarbeitet, das es dem Benutzer erlaubt, selbst solche Funktionen zu formulieren. Eine konkrete Syntax hierfür wird im Rahmen dieser Arbeit nicht vorgegeben, die notwendigen Vorbereitungen ermöglichen jedoch eine sehr

einfache Implementierung benutzerdefinierter Funktionen. Als einfache Vorstufe wurde ein Konzept für Namensassoziationen ähnlich den `LET`-Konstrukten in LISP spezifiziert und implementiert.

- **späte Bindung**
  Die Bindung eines Operators an einen Operatornamen war in **MILOS** statisch, da jedem Operatornamen nur ein Operator zugeordnet war. Programmiersprachen, die eine Operatorüberladung erlauben, binden ihre Operatoren immer dynamisch. Statische Bindung hat sich hier als unzulänglich erwiesen. Auch die vorliegende Spezifikation verwendet späte, also dynamische Bindung. Operatoren werden erst im Moment der Auswertung an das Operatorsymbol gebunden.

- **Übersichtlichkeit des Konzepts**
  Aufgrund der Operatorüberladung macht es keinen Sinn mehr, die Operatoren in der Spezifikation der Vorbedingungen aufzuführen. Dadurch wurde das gesamte Konzept einfacher und überschaubarer. Dies kommt vor allem zukünftigen Erweiterungen zugute.

Speziell die dynamische Bindung wird von Laien oft als schwer durchschaubar verstanden, da durch das Definieren (speziell das Überdefinieren) von Methoden sich das Auswertungsverhalten von Ausdrücken ändert, obwohl an den Ausdrücken selbst gar nichts geändert wurde. Statische Bindung, die bereits zur Übersetzungszeit vorgenommen wird, hat sich daher bei prozeduralen Programmiersprachen durchgesetzt. Speziell wenn Operatoren überladen werden können, treten aber auch bei statischer Bindung Effekte auf, die mitunter noch undurchschaubarer sind. Hier hängt das Ergebnis einer Auswertung nämlich möglicherweise davon ab, wann diese Auswertung formuliert (und dabei gebunden) wurde. D.h. zwei völlig identische Ausdrücke können bei gleichzeitiger Auswertung unterschiedliche Ergebnisse liefern. Daher wurde das alte dynamische Bindungskonzept der MILOS-Spezifikation ersetzt.

2.2 Die Spezifikation regulärer Ausdrücke

Abbildung 2 zeigt die Spezifikation einer Klassenhierarchie für reguläre Ausdrücke als StP-OMT-Diagramm. Genauere Erläuterungen zur Semantik der Darstellung von OMT-Diagrammen finden sich unter Anderem in [1].

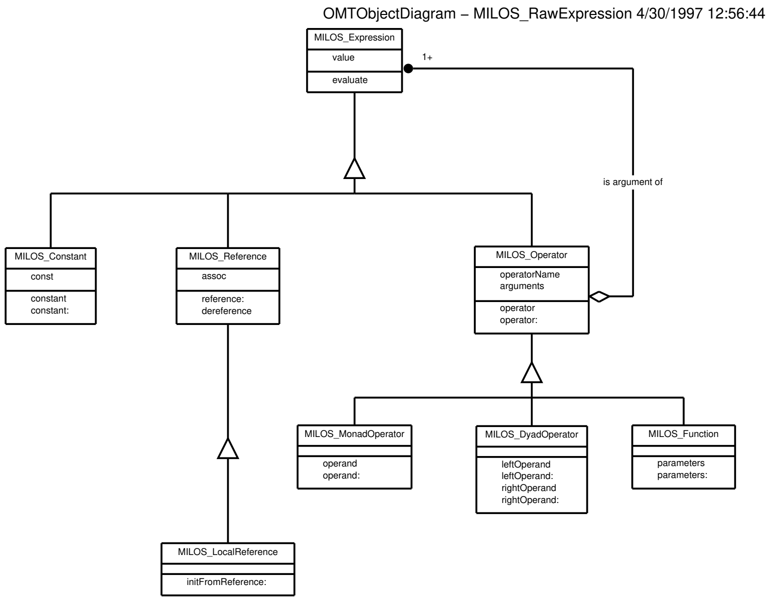

Abbildung 2: Klassenhierarchie für reguläre Ausdrücke

2.2.1 Ausdrücke

Ein Ausdruck wird repräsentiert durch eine Instanz der abstrakten Klasse `MILOS_Expression`. Diese wird zu drei Unterklassen verfeinert. Durch die Methode `evaluate` kann ein Ausdruck ausgewertet werden. Rückgabewert ist eine Instanz eines Produkttyps. Die Auswertung selbst wird durch die Unterklassen von `MILOS_Expression` vorgenommen. Durch die Methode `copy` kann eine tiefe Kopie eines Ausdrucks angelegt und dann mittels `becomeInstanceFor:` in eine Instanz verwandelt werden. Einige Ausdrücke sind nur als Instanzen auswertbar. Die Instantiierung wird durch die jeweilige Unterklasse vorgenommen.

2.2.2 Konstanten

`MILOS_Constant` repräsentiert eine lokale Konstante, die nur innerhalb dieses Ausdrucks sichtbar ist. Sie wird benötigt für einfache Vergleiche, wie "$x > 5$". Es ist jedoch auch möglich, komplexe Konstanten zu erzeugen. Die Zugriffsmethoden `constant` und `constant:` liefern bzw. erwarten Instanzen des **CoMo-Kit**-Typsystems. Eine Instantiierung ist nicht notwendig. Die Auswertung entspricht dem Zugriff auf die Konstante durch die Methode `const`.

2.2.3 Referenzen

`MILOS_Reference` stellt eine Referenz auf ein **CoMo-Kit**-Produkt dar. Durch die Zugriffsmethode `dereference` erhält man das referenzierte Objekt, durch `reference:` kann man die Referenz auf ein anderes Objekt umbiegen. Referenziert werden können Eingabeparameter der *Aufgabe* oder globale Konzeptinstanzen. Die Eingabeparameter sind durch den Datenfluß im Modell definiert. Es ist zu beachten, daß Eingabeparameter einer *Aufgabe* vor deren Instantiierung nur durch formale Parameter repräsentiert sind. Damit sind Referenzen auf diese Parameter nicht auswertbar. Erst durch die Instantiierung werden die Referenzen von einem formalen Parameter auf seine Instanz, einen aktuellen Parameter umgebogen. Die Auswertung geschieht durch dereferenzieren.

Als Verfeinerung existiert eine Klasse `MILOS_Localreference`. Diese stellt eine Referenz auf eine lokale Namensassoziation dar. Einziger Unterschied zur `MILOS_Reference` ist, daß bei der Auswertung nicht nur das referenzierte Objekt zurückgeliefert wird, sondern die Assoziation zuerst ausgewertet und das Ergebnis dieser Auswertung dann zurückgeliefert wird.

2.2.4 Operatoraufrufe

`MILOS_Operator` stellt keinen Operator, sondern einen Operatoraufruf dar. Die Operatoren selbst sind nicht Teil der regulären Ausdrücke. Sie werden auch von anderen Teilen des **CoMo-Kit**-Systems benötigt und im nächsten Kapitel spezifiziert.

`MILOS_Operator`
wird verfeinert zu den unären Operatoren (`MILOS_MonadOperator`), binären Operatoren (`MILOS_DyadOperator`) und Funktionen (`MILOS_Function`). Die Unterschiede dieser Verfeinerungen sind in erster Linie syntaktischer Natur und dienen der Übersichtlichkeit. Die Funktionsweise aller Operatoraufrufe ist identisch. Jeder Operator kennt seine Operanden. Bei der Auswertung wird ein geeigneter Operator gesucht und an den Operatoraufruf gebunden. Durch Auswertung dieses Operators mit den Operanden wird der Operatoraufruf dann selbst ausgewertet. Die Zugriffsmethoden unterscheiden sich wie folgt.

MILOS_MonadOperator operand, operand:

MILOS_DyadOperator leftOpd, leftOpd:, rightOpd, rightOpd:

MILOS_Function params, params:

Zudem wird die Klasse `MILOS_DyadOperator` verfeinert zur Klasse `MILOS_Selector`. Diese repräsentiert den Aufruf einen sogenannten Punkt-Operators (bekannt aus C++), der dazu dient auf ein Attribut einer Datenstruktur zuzugreifen. Realisiert wird er durch Aufruf eines Operators, der als Attribute die gesamte Struktur und den Namen des Attributs erwartet. Das verwendete Operatorkonzept stellt diese Funktionalität nicht zur Verfügung. Daher implementiert die Klasse `MILOS_Selector` nicht nur den Aufruf, sondern auch die Auswertung des Punkt-Operators. Zur Kapselung des Attributnamens wurde die Klasse `MILOS_Constant` verfeinert zu einer Klasse `MILOS_SlotIdentifier`. Ein `MILOS_SlotIdentifier` kapselt eine eindeutige Darstellung eines Attributnamens. Es sei erwähnt, daß die Funktionalität dieser beiden Klassen durchaus auch durch das Operatorkonzept hätte realisiert werden können. Dieser Weg wurde gewählt um eine anschaulichere Darstellung für den Benutzer zu gewährleisten. Ein Aufruf des Punkt-Operators wird z.B. nicht durch einen Ausdrucksbaum dargestellt, sondern durch einen einzigen Knoten der Form `struktur.attribut` wie aus C++, Pascal, etc. bekannt.

2.3 Erweiterung um Namensassoziationen

Abbildung 3 zeigt die nötigen Erweiterungen zu Behandlung von Namensassoziationen.

Eine Klasse `MILOS_LetExpression` kapselt den eigentlichen Ausdruck und beinhaltet zusätzlich eine geordnete Liste von Namensassoziationen. Dabei darf in jeder Assoziation eine bereits vorher definierte Assoziation referenziert werden. Folgendes Beispiel in der Syntax einer imperativen Sprache soll das erläutern:

Dieser reguläre Ausdruck würde bei der Auswertung die Zahl 3 ergeben

a := 1;
b := a+1;
return (b+1);

Der nun folgende Ausdruck ergibt eine Fehlermeldung bei (1). Der Name *b* ist hier noch nicht gebunden.

a := 1;
b := a + b; (1)
return (b+1);

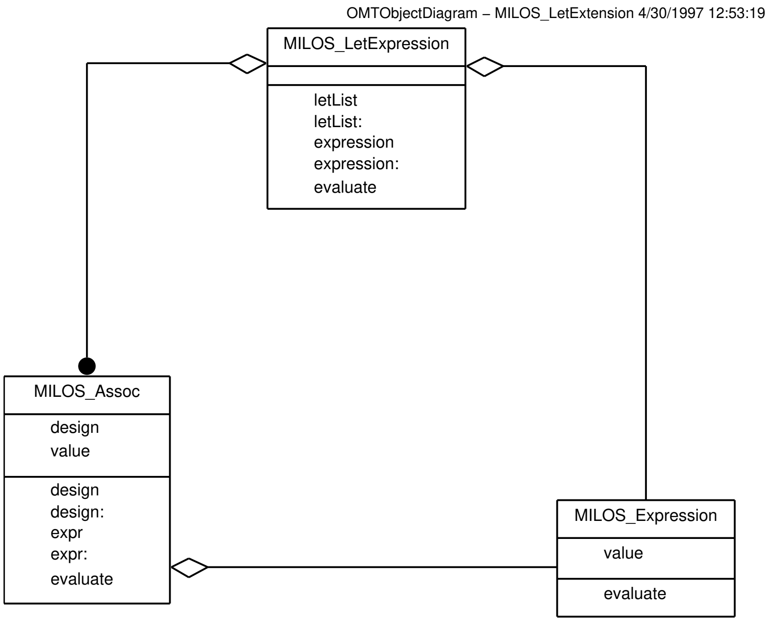

Abbildung 3: Erweiterung der Klassenhierarchie für Namensassoziationen

Dadurch sind zyklische Referenzen innerhalb von Namensassoziationen nicht möglich.

Die Klasse `MILOS_Assoc` stellt eine Assoziation dar. Dabei wird ein Ausdruck an einen Namen gebunden. Die Zugriffsmethoden `design` und `design:` erlauben den Namen zu erfragen bzw. zu ändern. Mit den Methoden `expression` und `expression` läßt sich der an den Namen gebundene Ausdruck erfragen bzw. ersetzen. Die Auswertung einer Namensassoziation geschieht durch Auswertung des angebundenen Ausdrucks. Die Instantiierung durch Instantiieren des Ausdrucks. Dabei sollten Assoziationen nicht direkt sondern nur über ihren Namen ausgewertet werden, d.h. nur wenn der reguläre Ausdruck zu dem die Assoziation gehört diese Assoziation benötigt wird sie tatsächlich ausgewertet. Dies entspricht einer späten Auswertung wie aus LISP bekannt.

2.4 Spezifikation des Operatorkonzepts

Da Operatoren auch von anderen Teil des **CoMo-Kit**-Systems benutzt werden, sind sie nicht Teil der `MILOS_Expression`-Hierarchie. Stattdessen werden sie in Form von Instanzen der Klasse `Operator` repräsentiert.

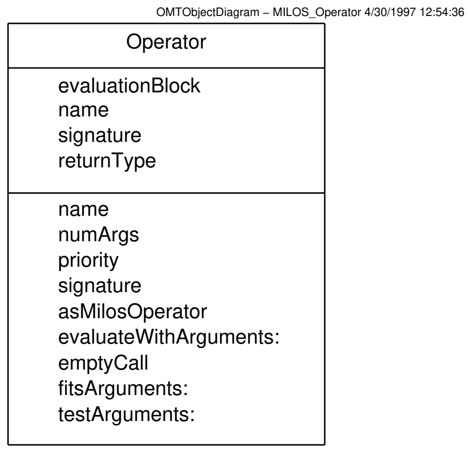

Abbildung 4: Die Klasse `Operator`

Eine Operator kennt seine Signatur in Form einer geordneten Liste von **CoMo-Kit**-Typen. Diese kann durch die Methode `signature` erfragt werden. Zudem kann durch die Methode `fitsArguments:` geprüft werden, ob eine Liste von Parametern zur Signatur des Operators paßt. Der Rückgabetyp wird nicht als Teil der Signatur behandelt, da sonst die Überladung nicht mehr eindeutig entscheidbar wäre. Stattdessen kann er über die Methode `returnType` erfragt werden. Durch die Methode `evaluateWithArguments:` kann schließlich der Operator ausgewertet werden. Als Eingabe erwartet er eine Liste von **CoMo-Kit**-Instanzen, Ergebnis ist eine Instanz von Typ `returnType`. Beim Aufruf eines Operators wird geprüft, ob alle Parameter mit definierten Werten belegt sind. Ist dies nicht der Fall, liefert der Operator ohne weitere Auswertung einen undefinierten Wert zurück. Allerdings gibt es Operatoraufrufe, bei denen trotz undefinierter Parameter der Rückgabewert definiert ist. Z.B. ergibt 'nil:FLOAT = nil:FLOAT' den Wert `TRUE` zurück. Um solche Operatoraufrufe behandeln zu können, verfügt jeder Operator über einen Testblock, der entscheidet, ob eine Auswertung stattfindet oder nicht. Der Standardtestblock bricht die Auswertung ab, sobald einer der Parameter einen undefinierten Wert enthält.

Die Überladung der Operatoren regelt die Klasse `Operator` selbst. Sie kennt alle ihre Instanzen, also alle zur Verfügung stehenden Operatoren. Auf die Methode `forArguments:` liefert die Klasse eine passende Instanz für die angegebene Argumentliste.

Die Methode `initializeAfterLoad` erzeugt eine Reihe von Standardoperatoren. Insbesondere werden alle Operatoren erzeugt, die in der alten **MILOS**-Spezifikation vorgesehen waren.

Es folgt eine Aufstellung der Standardoperatoren

Name	Signatur	Ergebnis	Beschreibung
$+$	`Float Float`	`Float`	arithmetische Addition
$-$	`Float Float`	`Float`	arithmetische Subtraktion
$*$	`Float Float`	`Float`	arithmetische Multiplikation
$/$	`Float Float`	`Float`	arithmetische Division
$-$	`Float`	`Float`	arithmetische Negation
div	`Float Float`	`Float`	arithmetische Ganzzahldivision
mod	`Float Float`	`Float`	Rest bei arithmetischer Ganzzahldivision
and	`Bool Bool`	`Bool`	logische Konjunktion
or	`Bool Bool`	`Bool`	logische Disjunktion
xor	`Bool Bool`	`Bool`	exklusive Disjunktion
not	`Bool`	`Bool`	logische Negation
$<$	`Float Float`	`Float`	kleiner
$<=$	`Float Float`	`Float`	kleiner oder gleich
$>$	`Float Float`	`Float`	größer
$>=$	`Float Float`	`Float`	größer oder gleich
$=$	`Concept Concept`	`Bool`	Identität
$<>$	`Concept Concept`	`Bool`	Negation der Identität

Der Typ `Concept` ist hierbei der Basistyp des Typsystems von **CoMoKit** und kann mit allen Typen instantiiert werden. Zusätzlich gibt es die Möglichkeit auf Attribute eines Konzepts zuzugreifen. Bei Programmiersprachen wie C++ wird das ebenfalls durch einen Operator, den sogenannten PunktOperator geleistet. Das hier vorgestellte Operatorkonzept enthält keinen solchen PunktOperator. Stattdessen wird diese Funktionalität von der bereits erläuterten Klasse `MILOS_Selector` implementiert.

3 Definition und Implementierung eines graphischen Interfaces

Zur Spezifikation der Vor- und Nachbedingungen ist nicht nur ein Konzept zur internen Darstellung vonnöten, es muß eine Eingabemöglichkeit zur Verfügung stehen um dem Benutzer die Formulierung solcher Vor- und Nachbedingungen zu ermöglichen. Dazu wurde der existierende Taskeditor des **CoMo-Kit**-Systems erweitert um ein graphisches Interface für Vor- und Nachbedingungen. Dieses Kapitel wird die Funktionsweise des Interfaces erläutern und auf Implementierungs- aspekte eingehen.

3.1 Der graphische Editor

Der **CoMo-Kit**-Taskeditor wurde im Menü Edit um einen Punkt Control erweitert. Abbildung 5 zeigt das Fenster des Taskeditors nach Anwahl des Menüpunktes Control.

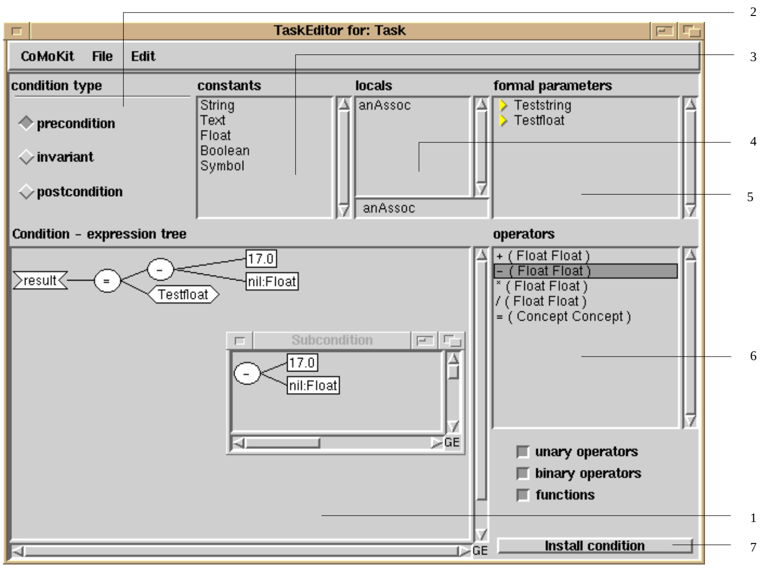

Abbildung 5: Die Benutzeroberfläche zu Eingabe von Vorbedingungen

Die Anzeige gliedert sich in ein großes Editierfeld links unten (1). Darüber befindet sich ein Auswahlfeld für dir Art der Bedingung (2) und Listen der zur Verfügung stehenden Typen (3), der definierten Namensassoziationen

(4) und der referenzierbaren **CoMo-Kit**-Parameter (5). Neben dem Anzeigefeld befindet sich eine Liste der verfügbaren Operatoren (6).

3.1.1 Anzeige- und Editierfeld

Im Anzeigefeld- und Editierfeld (1) wird die aktuelle Bedingung in Form eines Baumes dargestellt. Operatoren werden durch runde Symbole repräsentiert, Konstanten durch Rechtecke. Referenzen werden durch flache Sechsecke kenntlich gemacht. Sämtliche Objekte, die im Anzeigefeld dargestellt werden, können mit der linken Maustaste angewählt werden. Zur Erhöhung der Übersichtlichkeit wurde ein Konzept zum Wegfalten beliebiger Teilausdrücke implementiert. Durch einen Doppelclick mit der Maus wird der Teilbaum unterhalb des angewählten Objekts weggefaltet. Ein weiterer Doppelklick faltet den Teilbaum wieder auf. Die Abbildungen 6 und 7 zeigen die aufgeklappte und die zusammengefaltete Darstellung eines Teilausdrucks.

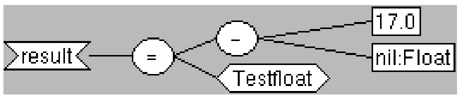

Abbildung 6: Aufgeklappter Teilausdruck

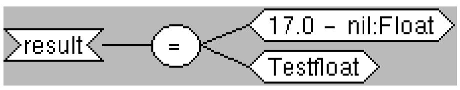

Abbildung 7: Zusammengefalteter Teilausdruck

Ist mindestens ein Objekt des Ausdrucksbaumes angewählt, steht durch Drücken der mittleren Maustaste ein Kontextmenü zur Verfügung. Folgende Operationen sind damit möglich:

Menüpunkt	Funktion
edit	öffnet ein neues Fenster mit einem neuen Anzeigefeld das nur den Teilbaum unterhalb des angewählten Objekts darstellt. Dadurch lassen sich Teile komplexer Bedingungen übersichtlich editieren.
set Value	öffnet einen Editor für die angewählte Konstante, in dem ihr Wert (bzw. mehrere Werte) geändert werden können. Nachstehende Abbildung 8 zeigt ein solches Editorfenster für `FLOAT`-Konstanten
replace with...	ermöglicht es, Teilbäume durch Konstanten der Basistypen zu ersetzen und ist zum schnellen Aufbau einfacher Operationen gedacht.
cut	schneidet den kompletten Baum unterhalb des gewählten Objekts ab und speichert ihn ihn einem internen Puffer, aus dem er später wieder an anderer Stelle in den Baum kopiert werden kann. Der abgeschnittene Zweig wird durch eine undefinierte Bool-Konstante ersetzt.
copy	legt eine Kopie des Teilbaumes unterhalb des angewählten Objekts im internen Puffer ab. Im Gegensatz zu cut bleibt der Teilbaum dabei unverändert.
paste	ersetzt den Teilbaum unterhalb des angewählten Objekts durch den Inhalt des internen Puffers. cut, copy und paste zeigen damit das bekannte verhalten gleichnamiger Befehle in Smalltalk.
compress	faltet den Teilbaum unterhalb des angewählten Objektes weg. Der gesamte Teilbaum wird durch ein einziges Objekt ersetzt, das diesen Teilbaum beschreibt. Der Befehl bewirkt damit dasselbe, wie ein Doppelclick mit der linken Maustaste.
expand	faltet einen zusammengeklappten Teilbaum wieder auf.
toggle	faltet aufgeklappte Teilbäume zusammen und zusammengeklappte Teilbäume auf.
inspect	erlaubt ein betrachten des angewählten Objekts.

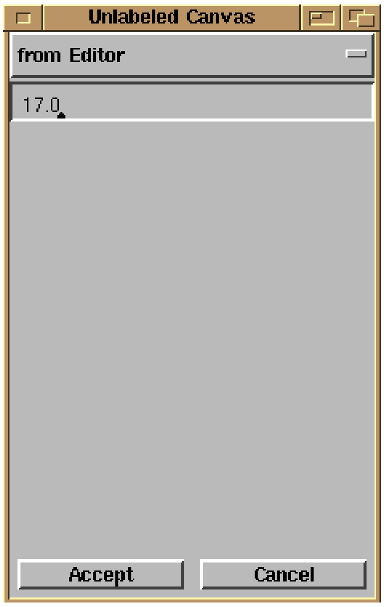

Abbildung 8: `FLOAT`-Editor

Durch diese Befehle lassen sich schon einfache Ausdrucksbäume erstellen. Komplexere Strukturen können in Verbindung mit den weiteren Elementen (2)-(6) der Oberfläche erzeugt werden und werden in den folgenden Abschnitten erläutert.

3.1.2 Auswahl der Bedingung

Der Editor ermöglicht das Formulieren von Vorbedingungen und Nachbedingungen. Durch Anwählen des entsprechenden Knopfes auf der Oberfläche wird die entsprechende Bedingung in den Editor geladen und im Anzeigefeld dargestellt. Zudem wird die Anzeige der referenzierbaren Objekte aktualisiert.

Wird eine Bedingung editiert, wird zunächst eine Kopie angelegt die dann verändert werden kann. Erst durch Drücken des Install-Knopfes (7) wird die alte Bedingung durch die geänderte Kopie ersetzt.

3.1.3 Liste der bekannten Typen

Da **CoMo-Kit** benutzerdefinierte Typen erlaubt, werden in diesem Feld alle bekannten Typen aufgelistet. Durch Anwahl eines Typs steht im Kontextmenü ein Punkt copy zur Verfügung, mit dem eine Instanz dieses Typs in den Internen Puffer kopiert werden kann. Dieser kann dann durch den Befehl paste in den Ausdrucksbaum eingefügt werden (siehe dort). Alternativ kann man eine Instanz auch per drag&drop einfügen. Dazu nimmt man eine Instanz des Typs mit der Maus auf und läßt sich auf ein zu ersetzendes Objekt im Ausdrucksbaum fallen. Wie beim paste-Befehl können nur Objekte des Baumes ersetzt werden. Es ist nicht möglich ein neues Objekt einfach auf das Anzeigefeld fallen zu lassen und später mit dem Ausdrucksbaum zu verbinden. Eine weitere Bedeutung der Typliste wird im folgenden Abschnitt beschrieben

3.1.4 Namensassoziationen

In diesem Feld stehen alle bekannten Namensassoziationen. Das Assoziationen nur lokal sichtbar sind, ist die Liste initial leer. Es gibt mehrere Möglichkeiten neue Assoziationen zu erzeugen. Zum einen gibt es unterhalb der Liste ein Eingabefeld für Strings. Gibt man dort einen Namen ein wird eine Assoziation unter diesem Namen erzeugt. Als Voreinstellung weist sie auf eine `Bool`-Konstante mit undefiniertem Wert. Ist im Typfeld ein anderer Typ angewählt, wird eine undefinierte Konstante dieses Typs als Bindung verwendet. Alternativ kann auch ein Typ per drag&drop aus der Typliste aufgenommen und auf die Assoziationsliste fallen gelassen werden. Assoziationen können im Kontextmenü editiert werden. Durch den Menüpunkt edit binding wird ein Anzeigefeld geöffnet, das die entsprechende Bindung als Baum darstellt. Diese kann dann genau wie im Feld (1) editiert werden. Um eine Assoziation im Ausdrucksbaum zu benutzen, kann eine lokale Referenz auf diese Assoziation mit dem Befehl copy im im internen Puffer angelegt werden und später in den Ausdrucksbaum eingefügt werden. Alternativ kann man auch hier die Assoziation per drag&drop aufnehmen und auf den Ausdrucksbaum fallen lassen.

3.1.5 Referenzierbare Objekte

Das Feld (5) stellt alle referenzierbaren Objekte in Form einer hierarchischen Liste dar. Auf oberster Ebene befinden sich die formalen Parameter der aktuellen *Aufgabe* (zu der der Editor gehört) und globale Produkte, die

CoMo-Kit zur Verfügung stellt. Durch Aufklappen eines Eintrags erhält man Zugriff auf die Attribute bzw. Teilprodukte eines Objektes. Dadurch ist ein einfacher Zugriff auf diese Elemente eines Objekts möglich. Durch den Befehl copy im Kontextmenü wird eine Referenz auf das angewählte Objekt in den internen Puffer kopiert und kann dann in den Ausdrucksbaum eingefügt werden. Alternativ kann man eine Referenz auch per drag&drop aufnehmen und auf ein Objekt des Ausdrucksbaums fallen lassen.

3.1.6 Operatoren

Eine Liste aller Operatoren steht im Feld (6) zur Verfügung. Die Menge der Angezeigten Operatoren läßt sich der Übersicht halber einschränken. Durch Abwahl der Knöpfe unterhalb der Operatorliste lassen sich einstellige, zweistellige und/oder mehrstellige Operatoren ausblenden. Durch den copy-Befehl im Kontextmenü wird ein Operatoraufruf für den entsprechenden Operator im internen Puffer abgelegt. Zudem werden undefinierte Instanzen aller benötigten Argumente erzeugt und an den Operatoraufruf gebunden. Durch kopieren des Operators `+(Float Float)` wird also ein Operatoraufruf für den Operator `+` erzeugt und zwei undefinierte `Float`-Objekte werden als Operanden an den Operatoraufruf gebunden. Durch den paste-Befehl kann der Operatoraufruf dann in den Ausdrucksbaum eingefügt werden. Alternativ kann auch hier drag&drop verwendet werden.

Hierbei sei erwähnt, daß nicht der Operator selbst sondern nur ein Operatoraufruf mit entsprechendem Namen in den Ausdrucksbaum eingefügt wird. Da die Argumente des Aufrufs verändert werden können und der Operatoraufruf dynamisch gebunden wird kann es passieren, daß tatsächlich ein anderer Operator ausgeführt wird, als der eigentlich eingefügte. Folgendes Beispiel macht das deutlich.

Man wähle den Operator `+(Float Float)` und kopiere ihn in den Ausdrucksbaum. Dann ersetze man die beiden `Float`-Argumente durch `Bool`'sche Argumente. Versucht man den Operator auszuwerten entsteht ein Laufzeitfehler: Es gibt keinen Operator `+`, der auf die Signatur `(Bool Bool)` paßt. Erzeugt man jedoch einen solchen, wird der Operatoraufruf korrekt abgearbeitet und der neu erzeugte Operator wird ausgewertet. Der ursprünglich eingefügte Operator `+(Float Float)` wird dabei nicht verwendet.

3.1.7 Untereditoren

Wird im Editorfeld (1) der Menüpunkt edit angewählt, öffnet sich ein neues Fenster mit einem Untereditor. Abbildung 9 zeigt ein solches Untereditorfenster.

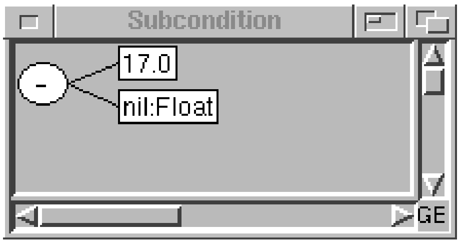

Abbildung 9: Editorfenster für einen Teilausdruck

Dieser Editor stellt einen Teilausdruck des gesamten Baumes da. Wurzel dieses Teilbaumes ist das Objekt, das im Haupteditor angewählt war. Die Funktionsweise dieses Untereditors ist identisch zum Editorfeld (1). Konstanten, Referenzen und Operatoren können per copy&paste oder per drag&drop in den Untereditor eingefügt werden. Teilbäume können zischen Untereditor, Haupteditor und ggfs. anderen Untereditoren per cut,copy&paste ausgetauscht werden. Zu beachten ist, daß beim Öffnen eines Untereditor immer eine Kopie des zu editierenden Teilbaumes erzeugt wird. Diese kann dann im Untereditor verändert werden, ohne daß andere Editoren beeinflußt werden. Im Kontextmenü steht ein Menüpunkt accept zur Verfügung. Dieser ersetzt den ursprünglichen Teilbaum durch die veränderte Kopie im Untereditor.

3.2 Implementation des Editors

Dieser Abschnitt soll die Implementation des graphischen Editors erläutern. Diese gliedert sich in drei Abschnitte. Der Erste Abschnitt besteht in der Erweiterung des **GraphEditor** von Frank Weberskirch. Dieser ist für die Verwaltung des Editorfeldes (1) zuständig und damit der Kern des gesamten Interfaces. Im Zweiten Teil wird des Interface selbst mit den Eingabefeldern (2)-(7) erläutert. Teil drei schließlich wird auf die Implementierung der in Kapitel 2 vorgestellten Spezifikation für Reguläre Ausdrücke eingehen. Dort wurden umfassende Erweiterungen der Spezifikation zur Unterstützung der graphischen Darstellung vorgenommen. Abbildung 10 zeigt die Spezifikation des Modells als OMT-Diagramm und die Einbettung der Klassenhierarchie aus Kapitel 2.

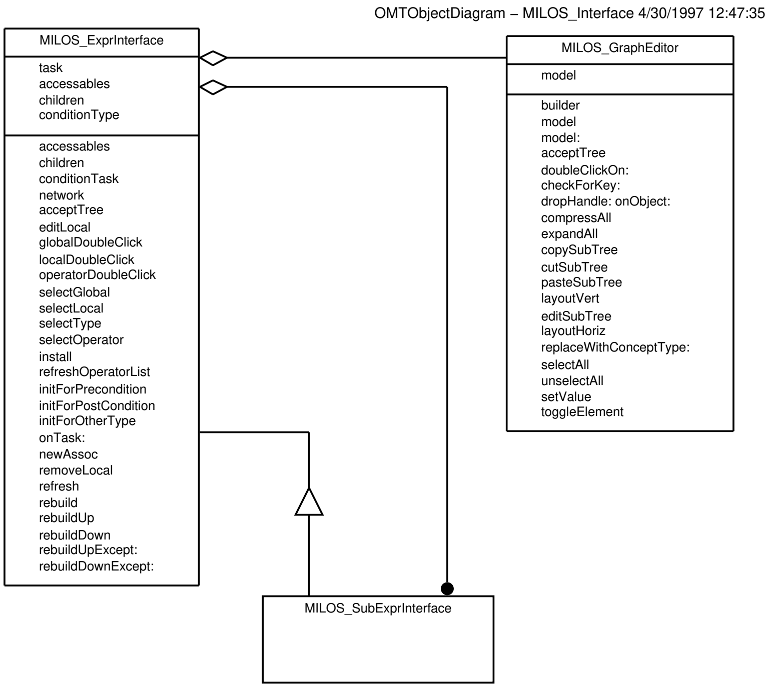

Abbildung 10: Modell der Benutzeroberfläche

3.2.1 Der MILOS_GraphEditor

Mit dem **GraphEditor** von Frank Weberskirch steht unter Smalltalk ein Werkzeug zur Verfügung, mit dem sich beliebige Graphen sehr einfach erzeugen, darstellen und editieren lassen. `MILOS_GraphEditor` verwendet den `TreeEditor`, eine Erweiterung des `GraphEditor`, die automatische Layouts speziell für Bäume implementiert. `MILOS_GraphEditor` ist jedoch keine Erweiterung des **GraphEditors**, sondern verfeinert die Klasse `GEApplication`, welche den **GraphEditor** kapselt.

Um einen Baum im `GraphEditor` darzustellen wird das Wurzelobjekt diese Baumes an den `GraphEditor` übergeben und eine Methode, mit der der `GraphEditor` die Nachfolger eines Knotens und insbesondere der Wurzel in Erfahrung bringen kann. Damit Baut der `GraphEditor` dann einen internen Baum aus Objekten der Klasse `GraphNode` auf, wobei jedes dieser

Objekte genau einem Objekt des ursprünglichen Baumes zugehörig ist. Der `GraphEditor` legt also eine interne Kopie des Originalbaumes an, auf der er dann arbeiten kann. Die Knoten dieser Kopie kennen ihre zugehörigen "Originale", so daß Änderungen an der Kopie dem "Original" bekannt gemacht werden können. Die Behandlung solcher Änderungsnachrichten und die korrekte Durchführung der Änderungen ist Aufgabe des Benutzers, in diesem Fall also der Klasse `MILOS_GraphEditor`.

Im Folgenden sollen die Erweiterungen der Klasse `MILOS_GraphEditor` kurz erläutert werden.

Methodenname	**Funktion**
compressAll	faltet alle Objekte des Baums zusammen
expandAll	faltet alle zusammengeklappten Elemente auf
copySubTree	kopiert den angewählten Teilbaum in den internen Puffer
cutSubTree	schneidet den angewählten Teilbaum aus und kopiert ihn in den internen Puffer
pasteSubTree	ersetzt das angewählte Element durch den Inhalt des internen Puffers
inspectElement	öffnet für alle angewählten Elemente ein inspect-Fenster
selectAll	wählt alle aufgeklappten Elemente des Ausdrucksbaumes an
unselectAll	wählt alle Elemente ab
setValue	öffnet ein Editorfenster für das angewählte Element des Baumes.
toggleElement	faltet ein angewähltes Element weg bzw. auf
replaceWithConceptType:	ersetzt das angewählte Objekt durch eine undefinierte Instanz eines **CoMo-Kit**-Konzepts. Als Frontend stehen die Methoden `replaceWithFloat`, `replaceWithSymbol`, `replaceWithBoolean` und `replaceWithString` zur Verfügung

Neben diesen existieren noch Zugriffsmethoden `model` und `model:`, über die das darüberliegende `MILOS_ExprInterface` erfragt bzw. geändert werden kann. Das `MILOS_ExprInterface` wird zwei Abschnitte weiter näher beschrieben. Die Klasse `MILOS_GraphEditor` stellt über die Methoden `PasteBuffer` bzw. `PasteBuffer:` Zugriff auf den internen Puffer zur Verfügung.

3.2.2 Erweiterungen der `MILOS_Expression`-Hierarchie

Für die "Zusammenarbeit" mit dem `MILOS_GraphEditor` war es notwendig, die in Kapitel 2 spezifizierten Klassen um diverse Funktionen zu erweitern. Abbildung 11 zeigt zunächst die komplette Klassenhierarchie. Das dargestellte UIL-Diagramm wurde direkt aus Smalltalk erzeugt und enthält die kompletten Klassenbeschreibungen.

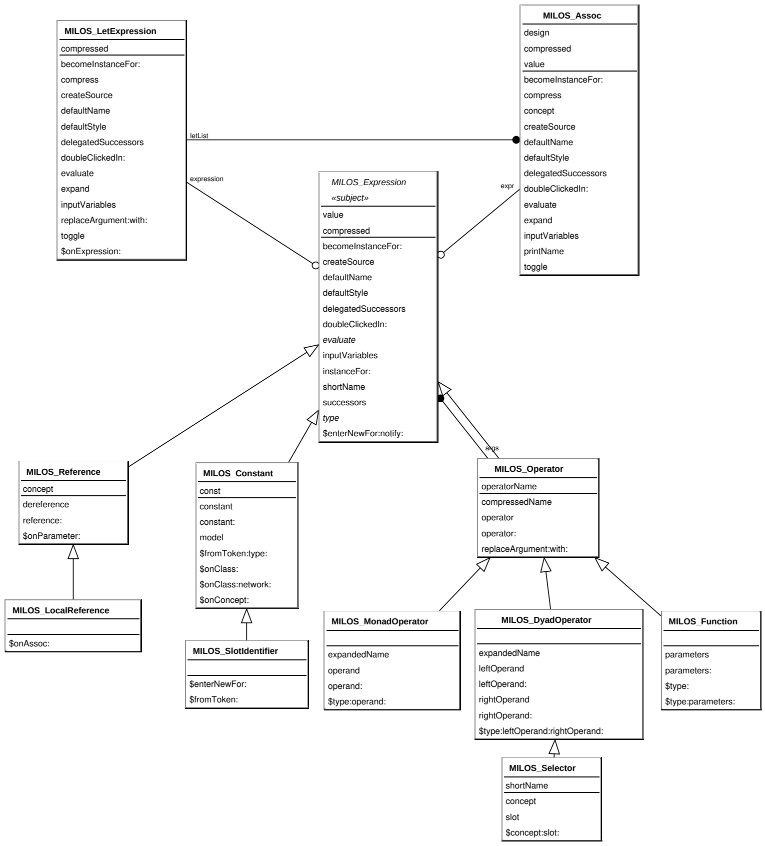

Abbildung 11: Vollständige Klassenhierarchie für reguläre Ausdrücke

Da die meisten neuen Methoden sehr einfach sind, sollen sie nur kurz und in tabellarischer Form erläutert werden.

Methodenname	**Funktion**
`delegatedSuccessors`	Wie eingangs erläutern übergibt man dem GraphEditor die Wurzel eines Baumes, den man editieren bzw. darstellen will und Zudem ein Methodenname , über den der GraphEditor die Kinder eines Baumknoten erhalten kann. Da das Interface das Teilweise Wegfalten von Ausdrücken erlaubt, muß diese Methode auch den Faltzustand eines Knotens kennen und entsprechend keine Kinderknoten zurückliefern, falls der Knoten selbst weggefaltet ist. Die Methode `delegatedSuccessors` leistet genau dies.
`successors`	Diese Methode liefert die Kinder eines Knotens ohne Rücksicht auf dessen Faltzustand. Sie wird von `delegatedSuccessors` verwendet.
`inputVariables`	Diese Methode liefert alle Eingabeparameter, die ein Baumknoten kennt. Sie entspricht damit der Methode `successors`.
`replaceArgument:` `with:`	ersetzt einen Operanden, also einen im ersten Argument angegebenen Kindknoten durch das zweite Argument. Diese Methode realisiert damit das ersetzen von Teilbäumen durch die im vorherigen Abschnitt beschriebene paste-Funktion. Sie wird vom `MILOS_GraphEditor` verwendet.
`doubleClickedIn:`	Diese Methode benachrichtigt einen Knoten davon, daß der Benutzer einen Doppelclick im GraphEditor auf diesem Objekt ausgeführt hat. Die Standard-Reaktion besteht im Auf- bzw. Wegfalten des Teilbaumes.
`defaultName`	Die Standardmethode liefert den Namen, unter dem ein Knoten im `GraphEditor` angezeigt wird. Üblicherweise entspricht dieser Name dem `displayString` eines Knotens.
`defaultStyle`	Liefert den Darstellungsstil im `GraphEditor` in Form eines `Symbol`. Eine Liste der erlaubten Stile kann über den `GraphEditor` erfragt werden.

`compress`
`expand`
`toggle`
	Diese Methoden ermöglichen die Steuerung des Faltzustandes eines Knotens.

3.2.3 Die Klasse MILOS_ExprInterface

Dies Klasse `MILOS_ExprInterface` realisiert das gesamte Eingabeinterface, wie es vom `TaskEditor` aufgerufen wird. Neben der Einbettung des `GraphEditor` steuert es alle am Anfang des Kapitels beschriebenen Eingabekomponenten. Die meisten Methoden wurden durch den VisualWorks-UIBuilder automatisch erzeugt. Daher sollen sie hier auch nicht näher erläutert werden. Die folgende Tabelle fast die wichtigeren und nicht automatisch generierten Methoden kurz zusammen.

Methode	**Funktion**
`accessables`	Liefert eine liste aller zugreifbaren, d.h. referenzierbaren **CoMo-Kit**-Objekte.
`children`	liefert eine Liste aller direkten Untereditoren.
`conditionTask`	liefert die `TaskDescription`, zu der die aktuelle Bedingung gehört
`network`	liefert das `HAM_Network`, in dem sich der `conditionTask` befindet und die referenzierbaren Objekte `accessables` befinden.
`acceptTree`	propagiert den aktuellen Zustand des Baumes an alle Untereditoren und erzeugt einen konsistenten Gesamtzustand. Nicht propagierte Änderungen in Untereditoren gehen dabei verloren.
`editLocal`	öffnet einen Untereditor für die aktuell angewählte Namensbindung in der Liste für lokale Namensassoziationen.
`globalDoubleClick`	behandelt einen Doppelclick mit der Maus in der Liste der referenzierbaren Objekte. Dabei wird eine Referenz auf den angeklickten Operator erzeugt und im internen Puffer abgelegt. Diese Referenz kann dann per paste-Befehl in den Ausdrucksbaum eingefügt werden.

`localDoubleClick` behandelt einen Doppelclick in die Liste der Namensassoziationen. Eine Referenz auf den angeklickten Namen wird im internen Puffer abgelegt und kann per paste in den Ausdrucksbaum kopiert werden.

`operatorDoubleClick` behandelt einen Doppelclick in die Liste der Operatoren. Es wird Operatoraufruf mitsamt den benötigten Operanden erzeugt und im internen Puffer abgelegt. Näheres hierzu findet sich in der Funktionsbeschreibung der Oberfläche.

`selectGlobal` behandelt eine einfachen Mausklick in die Liste der referenzierbaren Objekte. Als Standard wird keine besondere Aktion ausgelöst.

`selectLocal` behandelt einen einfachen Mausklick in die Liste der lokalen Namensassoziationen. Als Standard wird keine besondere Aktion ausgelöst.

`selectOperator` behandelt einen einfachen Mausklick in die Liste der Operatoren. Als Standard wird keine besondere Aktion ausgeführt.

`selectType` behandelt einen einfachen Mausklick in die Liste der bekannte **CoMo-Kit**-Typen. Der angewählte Typ wird dann als Vorgabetyp bei der Erzeugung neuer Namensassoziationen verwendet.

`refreshOperatorList` initialisiert die Anzeigeliste der Verfügbaren Operatoren. Diese Methode wird immer dann aufgerufen, wenn der Benutzer die Anzeigefilter der Operatoren verändert hat. Näheres findet sich in der Funktionsbeschreibung der Oberfläche.

`install` Diese Methode installiert den Ausdrucksbaum als Vorbedingung.

`newAssoc` behandelt das Anlegen einer neues Namensassoziation durch den Benutzer. Die Liste der Assoziationen wird neu initialisiert.

`removeLocal`	behandelt das entfernen einer Namensassoziation durch den Benutzer. Die Liste der Namensassoziationen wird initialisiert. Referenzen auf den gelöschten Namen werden durch undefinierte Konstanten desselben Typs ersetzt.

`refresh`
`rebuild`
`rebuildUp`
`rebuildUpExcept:`
`rebuildDown`
`rebuildDownExcept`
	Diese Methoden bauen den gesamten Ausdrucksbaum neu auf, sie übergeben das Wurzelobjekt an den `GraphEditor` und veranlassen ihn, eine neue interne Kopie des Baumes anzulegen. Nach jeder Änderung des Ausdrucksbaumes muß eine dieser Methoden aufgerufen werden um diese Änderung dem `GraphEditor` bekannt zu machen. Im Anhang an diese Tabelle wird die genaue Funktion der einzelnen Methoden erläutert.

`onTask:`	initialisiert das Interface für die angegebene Aufgabe. Die Vorbedingung der Aufgabe wird kopiert, die Kopie wird in der Editor geladen und dargestellt. Die Aufgabe (das Argument) wird gemerkt und kann durch die Methode `conditionTask` erfragt werden.
`initForPrecondition`	spezielle Initialisierungen für Vorbedingungen werden hier vorgenommen. Insbesondere wird die Kopie der Vorbedingung hier angelegt. Die Methode wird auch von `onTask:` aufgerufen.
`initForPostcondition`	hier werden Initialisierungen für die Behandlung von Nachbedingungen vorgenommen. Insbesondere wird eine Kopie der aktuellen Nachbedingung des `conditionTask` angelegt.
`initForOtherType`	Beim Umschalten von Vor- zu Nachbedingungen sind einige Maßnahmen nötig. Diese Methode tut genau das.

`updateLocals`	Diese Methode wird immer dann aufgerufen, wenn die Liste der lokalen Namensassoziationen geändert wurde. Sie regelt den Neuaufbau der Anzeige und gibt Änderungen an den GraphEditor weiter. Sie wird u.A. von `newAssoc` und `removeLocal` aufgerufen.

Die Methoden `refresh`, `rebuild`, `rebuildUp`, `rebuildUpExcept:`, `rebuildDown` und `rebuildDownExcept:`. Dienen dem Neuaufbau der Bildschirmdarstellung des `GraphEditor`. Insbesondere werden sie benötigt um Änderungen in Untereditoren an andere Editoren zu propagieren und dort ebenfalls einen Neuaufbau der graphischen Anzeige auszulösen. Die Methode `refresh` führt zunächst einfach eine neue Darstellung des Graphen durch. Sie wird von den anderen Methoden aufgerufen. `rebuild` hingegen baut den internen Baum des `GraphEditor` neu auf. Es wird eine komplett neue Kopie des Ausdrucksbaumes im `GraphEditor` erzeugt und danach dargestellt. Diese Methode muß immer dann aufgerufen werden, wenn eine Änderung des Original Ausdrucksbaumes durchgeführt wurde, die dem `GraphEditor` bekannt gemacht werden soll. Dies ist z.B. beim Ausschneiden von Teilbäumen durch den `cut`-Befehl notwendig. Änderungen, die nicht die Struktur des Ausdrucksbaumes betreffen, z.B. Änderung des Wertes einer lokalen Konstanten, erfordern keinen Neuaufbau des Baumes. Hier reicht der Aufruf der Methode `refresh`. Wird ein Teilausdruck in einem Editor geändert, so beeinflußt das evtl. auch andere Unter- bzw. Vatereditoren, die denselben Teilausdruck darstellen. Eine solche Änderung zu allen anderen Editoren zu propagieren ist Aufgabe der Methoden `rebuildUp`, `rebuildUpExcept:`, `rebuildDown` und `rebuildDownExcept:`. Durch `rebuildDown` wird zunächst ein Neuaufbau des Baumes veranlaßt und dann ein Neuaufbau aller Untereditoren durch rekursiven Aufruf der Methode `rebuildDown`. Ist der aufrufende Editor selbst ein Kindeditor, d.h. besitzt er Väter, so wird auch "nach oben hin" ein Neuaufbau durch rekursiven Aufruf der Methode `rebuildUp` durchgeführt. An der Wurzel, d.h. beim obersten Vatereditor angelangt wird dann ein Neuaufbau von dessen Kindeditoren veranlaßt durch rekursiven Aufruf der Methode `rebuildDown`. Durch diesen Mechanismus kann es zu endlosen rekursiven Aufrufen zwischen Vater und Söhnen kommen. Daher gibt es durch die Methoden `rebuildDownExcept:` und `rebuildUpExcept:` die Möglichkeit Ausnahmen zu formulieren. Diese Ausnahmen werden nicht mehr neu aufgebaut. Um zu verhindern, daß ein Editor unendlich oft neu aufgebaut wird, behandelt er sich selbst also als solche Ausnahme und verhindert so, daß er erneut refresht wird. Abbildung 12 veranschaulicht das Konzept.

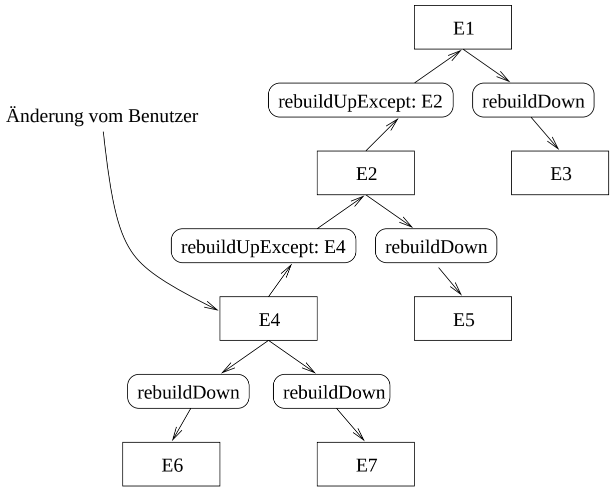

Abbildung 12: Notifikationsmechanismus der Teileditoren

3.3 Sonstige Erweiterungen

Zur Anbindung an das bestehende **CoMo-Kit**-System waren einige Änderungen an bestehenden Klassen nötig. Zudem mußten neue Klassen definiert werden, die der Anbindung an das Modell dienen und in erster Linie technische Natur sind. Daher sollen diese hier nur der Vollständigkeit halber kurz erwähnt werden.

- Das Edit-Menü des `TaskEditor` wurde um einen Punkt Conditions erweitert. Dieser Menüpunkt ruft den oben beschriebenen Editor auf.
- Eine Klasse `MILOS_Condition` wurde implementiert, die die eigentliche Vor- bzw. Nachbedingung kapselt. Sie wurde in den `TaskEditor` eingefügt und erlaubt externen Zugriff auf die Vor- und Nachbedingungen. Zudem regelt sie die Instantiierung einer Bedingung (s. Kapitel 4).
- Eine Klasse `MILOS_ExpressionFilter` wurde implementiert. Sie dient dazu unerwünschte Operationen auf Teile des Ausdrucksbaumes im GraphEditor abzufangen.

- Die **CoMo-Kit**-Konzepte im Form der Klasse `Concept` wurden erweitert um eine Methode `createMilosSource`. Diese Methode erzeugt eine kompakte String-Repräsentation eines Konzepts, so daß es im `GraphEditor` dargestellt werden kann.

4 Anbindung an den Scheduler

4.1 Instantiierung der Vorbedingung

Die Vorbedingung einer Aufgabe wird zunächst im Modell formuliert und bei der Aufgabenbeschreibung abgelegt. Zu diesem Zeitpunkt sind die formalen Parameter, die in der Vorbedingung referenziert werden können, noch nicht mit Werten belegt. Zudem sind die Operatoren noch nicht gebunden. Genau wie bei den Aufgabenmodellen selbst ist daher eine Instantiierung der Vorbedingung nötig. Erst dann ist die Vorbedingung auswertbar. Zu diesem Zweck wurde die bereits erwähnte Klasse `MILOS_Condition` um eine Methode `instanceFor:` erweitert. Nachdem die Aufgabenbeschreibung selbst instantiiert ist und damit auch ihre aktuellen Parameter kennt, wird diese Methode aufgerufen. Als Argument erwartet sie die Instanz der Aufgabenbeschreibung. Durch diese erfährt sie die aktuellen Parameter der Aufgabe, also die Bindung der formalen Parameter.

Die Methode `instanceFor:` legt zunächst eine Kopie der Vorbedingung an und instantiiert deren Teilausdrücke durch einen rekursiven Aufruf der Methode `becomeInstanceFor:`. Diese wiederum erzeugt Bindungen für die Operatoren und Instantiiert die Referenzen, d.h. die Referenzen auf Formale Parameter werden auf aktuelle Parameter umgebogen. Dabei sind aktuelle Parameter die Belegungen (Bindungen) der formalen Parameter, also die Bindungen Nach diesem Aufruf ist die so erzeugte Instanz der Vorbedingung auswertbar und kann der Aufgabe übergeben werden. Durch Aufruf der Methode `evaluate` kann dann die Auswertung der Vorbedingung, genauer der Instanz einer Vorbedingung, ausgelöst werden. Zu beachten ist, daß bei der Instantiierung die Klasse `MILOS_Condition` nicht mitinstantiiert wird. Sie ist nur Bestandteil des Modells. Eine Instanz ist also immer ein Objekt der Klasse `MILOS_LetExpression`, die im Modell in der `MILOS_Condition` gekapselt ist. An diese Instanz muß auch die Methode `evaluate` geschickt werden.

4.2 Verwaltung von Abhängigkeiten

Die Abwicklung eines **CoMo-Kit**-Modells wird durch einen Scheduler organisiert. Mit Hilfe eines TMS propagiert dieser Änderungen und verwaltet Abhängigkeiten. Eine genaue Beschreibung zu diesem Konzept findet sich in [2].

Ob eine Aufgabe gerade ausgeführt werden kann, wird durch einen TMS-Knoten `executable` repräsentiert. Ist dieser Knoten gerechtfertigt, kann die zugehörige Aufgabe abgewickelt werden. Die Rechtfertigung bestand bisher in einer konjunktiven Verknüpfung von TMS-Knoten `assigned`, welche die korrekte Belegung einer Eingabevariablen repräsentieren. Eine Aufgabe ist dann ausführbar, wenn alle Eingabeparameter korrekte Bindungen haben. Abbildung 13 stellt diese Abhängigkeit graphisch dar.

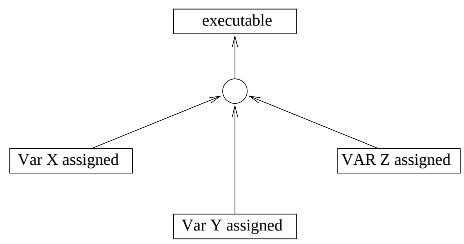

Abbildung 13: Altes Abhängigkeitsmodell

4.2.1 Änderungen der Abhängigkeiten durch die neue Spezifikation

Die in dieser Projektarbeit spezifizierten expliziten Vorbedingungen sind ausdrucksstark genug, die alten Bedingungen (Parameter sind belegt) vollständig zu ersetzen. Durch einen Operator exists lassen sich äquivalente Bedingungen auch explizit ausdrücken. Damit ließe sich das Prinzip der Datenflußsteuerung, das bisher verfolgt wurde, komplett ersetzen. Da es für den Anwender jedoch deutlich übersichtlicher und durchschaubarer ist, wurde dieses Prinzip beibehalten. Demnach ist eine Aufgabe genau dann ausführbar, wenn alle Eingabeparameter belegt sind **und** die explizite Vorbedingung erfüllt ist. Die Vorbedingung stellt also nur eine Verschärfung der alten impliziten Ausführungsbedingung zur Verfügung.

In Kapitel 1 wurde gezeigt, daß auch eine Aufweichung der Bedingung notwendig sein kann. Hierzu stehen dem Benutzer optionale Parameter zur

Verfügung. Diese optionalen Parameter sind nicht notwendigerweise belegt und werden auch nicht zur Steuerung der Ausführbarkeit verwendet. Demnach lassen sich beliebige Vorbedingungen realisieren durch optional Machen aller Eingangsparameter und Eingabe expliziter Vorbedingungen auf diesen Parametern. Damit steht die Volle Ausdruckskraft zur Verfügung, ohne auf das Prinzip der Datenflußsteuerung zu verzichten. Allerdings wird dieses Prinzip durch die Verwendung optionaler Parameter umgangen.

Zur Steuerung der Ausführbarkeit bietet sich die Verwendung eines TMS-Knotens `preconditionSatisfied` an. Dieser Knoten stellt eine zusätzliche Rechtfertigung zum `executable`-Knoten einer Aufgabe dar, d.h. die Aufgabe kann erst abgearbeitet werden, wenn auch dieser Knoten gerechtfertigt ist. Die Rechtfertigung geschieht dabei durch Auswertung der Bedingung auf Smalltalk-Ebene. Die Gültigkeit einer Vorbedingung und damit auch ihres Knotens hängt von den Belegungen der Eingabeparameter ab. Diese A andern sich zur Laufzeit. Um diese Abhängigkeit zu modellieren wurde ein weiterer TMS-Knoten `preconditionChanged` in das TMS eingefügt. Dieser Knoten ist durch die Belegung der Eingabeparameter gerechtfertigt und dient zur Notifikation des `preconditionSatisfied`-Knotens.

Abbildung 14 zeigt die neuen Abhängigkeitsstrukturen. Zur Verdeutlichung ist der neue Knoten `preconditionSatisfied` und seine Rechtfertigung fett dargestellt.

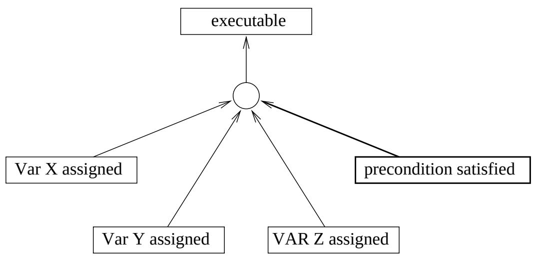

Abbildung 14: Neues Abhängigkeitsmodell

4.2.2 Spezifikation der neuen TMS-Strukturen

Die in vorherigen Abschnitt beschriebenen Abhängigkeiten müssen dem Scheduler bekannt gemacht werden. Das geschieht durch das Erzeugen von TMS-Knoten und das Einfügen dieser Knoten in das TMS-Netzwerk. Zunächst sind zwei neue Knoten Notwendig.

preconditionChanged

Methode	Funktion
`evalNotification`	Wird aufgerufen, sobald eine Variablenbindung geändert wurde. Die Notifikation wird an den Knoten `preconditionSatisfied` weitergeleitet.
`satisfiedNode`	liefert den zugehörigen `preconditionSatisfied` - Knoten.
`satisfiedNode:`	erlaubt das setzen des zugehörigen `preconditionSatisfied` - Knotens.

preconditionSatisfied repräsentiert die Erfülltheit einer Vorbedingung. Dieser Knoten ersetzt die Rechtfertigung des `executable`-Knotens durch die Belegung der Eingabeparameter. Die Rechtfertigung des Knotens geschieht durch deine Prämisse, die nach Auswertung der Vorbedingung auf Smalltalk-Ebene gesetzt wird.

preconditionChanged repräsentiert die Abhängigkeit der Vorbedingung von den Belegungen der Eingabevariablen. Dieser TMS-Knoten wird durch die Variablenbindungen gerechtfertigt, die ursprünglich die Ausführbarkeit einer Aufgabe gerechtfertigt hatten. Er ändert sich sobald eine Variablenbelegung geändert wird und veranlaßt, daß der `preconditionSatisfied` - Knoten erneut die Vorbedingung auswertet und seine Prämisse neu setzt bzw. entfernt. Die Benachrichtigung geschieht nicht auf TMS-Ebene, sondern auf Smalltalk- Ebene durch einen gewöhnlichen Notifikationsmechanismus.

Diese neuen TMS-Knoten wurden durch Verfeinerung der Klasse `CoMo-Kit_GoalNode` erzeugt.

4.3 Anbindung an das TMS

Die so spezifizierten TMS-Knoten müssen vor der Abwicklung des Modells erzeugt und in das TMS-Netzwerk eingefügt werden. Dazu wurde die Methode `buildNodesAndDependencies` erweitert. Es wird je ein Knoten `preconditionSatisfied` und ein Knoten `preconditionChanged` erzeugt. Beide Knoten werden in das Netzwerk integriert, so daß die oben spezifizierten

preconditionSatisfied

Methode	Funktion
`evalNotification`	wird nur durch den Knoten `preconditionChanged` aufgerufen. Die Instanz der Vorbedingung wird neu ausgewertet. Die Prämisse des Knotens wird Anhand des Ergebnisses dieser Auswertung gesetzt oder entfernt.

Abhängigkeiten korrekt repräsentiert sind. Die Weitere Verwaltung übernimmt der Scheduler.

4.4 Nachbedingungen

Da die Nachbedingungen für eine Aufgabe den Vorbedingungen sehr ähnlich sind, wurden sie bisher nicht explizit spezifiziert. Ihre Spezifikation entspricht tatsächlich exakt der der Vorbedingungen, bis auf die folgenden Änderungen.

- Eine `MILOS_Condition` für die Nachbedingung wird angelegt und unter dem Namen postcondition zur Aufgabe hinzugefügt. Die Aufgabenbeschreibung muß um Zugriffsmethoden `postcondition` und `postcondition:` zum erfragen bzw. setzen dieser Bedingung erweitert werden.
- TMS-Knoten `postconditionChanged` und `postconditionSatisfied` müssen erzeugt werden. Sie sind im Aufbau identisch zu den entsprechenden Knoten für Vorbedingungen. `postconditionSatisfied` rechtfertigt jedoch nicht den `executable`-Knoten einer Aufgabe, sondern den `completed`-Knoten.

Mit diesen Änderungen können auch Nachbedingungen operationalisiert werden.

4.5 Invarianten

Neben Vor- und Nachbedingengen erscheint es auch sinnvoll Invarianten zu formulieren und bei der Prozeßabwicklung zu berücksichtigen. Ähnlich wie bei den Nachbedingungen könnte die Interfaces sehr einfach dahingehend erweitern, daß der Benutzer selbst Invarianten formulieren und an eine Aufgabe hängen kann. Allerdings ist die Einbindung in den Scheduler noch etwas problematisch: Die Semantik von Vorbedingung und Invariante sich überschneidet. In der momentanen Implementierung ist eine Vorbedingung automatisch auch Invariante ihrer Aufgabe. Da **CoMo-Kit** keine Referenzparameter kennt, erscheint es sogar schwierig überhaupt einen Unterschied zwischen Vorbedingung und Invariante festzustellen. Aus diesem Grund wurde in dieser Projektarbeit von einer Behandlung von Invarianten abgesehen.

4.6 Ausblick: Bedingungen für Planung und Abwicklung

Die hier vorgestellten Vor- und Nachbedingungen beziehen sich immer auf die Abwicklung einer Aufgabe. Wie in [4] spezifiziert, könnte es aber auch sinnvoll sein, insbesondere Vorbedingungen zur Planung zu formulieren, d.h. Bedingungen die erfüllt sein müssen, bevor eine Aufgabe delegiert werden kann. Allerdings wäre dazu eine völlig andere Anbindung nötig, wobei das verwendete TMS-Netz noch gar nicht die notwendigen Strukturen zur Verfügung stellt. Daher geht diese Projektarbeit nicht weiter auf Vorbedingungen zur Planung ein, näheres findet sich jedoch in [4].

4.7 Zusammenfassung

In dieser Projektarbeit wurde ein Konzept zur Formulierung und Abarbeitung von Vor- und Nachbedingungen in **CoMo-Kit** Prozessen vorgestellt. Mit diesem Konzept ist es möglich die Abarbeitung von **CoMo-Kit** Prozessen explizit zu steuern und komplexere Zusammenhänge zu formulieren, als durch die vorher verwendeten Datenflußabhängigkeiten möglich war. Um das **CoMo-Kit** System für den Anwender durchschaubar zu halten, bleiben die Datenflußabhängigkeiten erhalten, können jedoch beliebig abgeschwächt und durch explizite Bedingungen verändert werden. Eine im Rahmen dieser Projektarbeit implementierte Benutzeroberfläche erlaubt es auf komfortable Art und Weise Vorbedingungen zu formulieren und zu ändern.

Literatur

[1] Kötting B.: *Konzeption und Implementierung einer Sprache für Software-Entwicklungsprozesse*. Diplomarbeit, Universität Kaiserslautern, 1995.

[2] Dellen B.: *Zustandsmodelle für flexible Entwurfsprozesse*. Diplomarbeit, Universität Kaiserslautern, 1995.

[3] Dellen B., Maurer F., Paulokat J.: *Verwaltung von Abhängigkeiten in kooperativen wissensbasierten Arbeitsabläufen*, In M.M. Richter und F. Maurer (Hrsg.), Proceedings in Artificial Intelligence 2, Expertensystem 95, 1995.

[4] Christofidelli S.: *Spezifikation einer flexiblen Abwicklungsmaschine für Workflowprozesse*. Projektarbeit, Universität Kaiserslautern, 1997.

[5] Goldberg A.: *SMALLTALK 80 - The Language and its implementation*. Addison Wesley, 1983.