Generierung und Erweiterung des Resource Flow Graph für Visual Basic 6 Programme

Diplomarbeit

Jan Harder

Matrikelnummer: 1358811

11.12.2006

Universität Bremen

Fachbereich Mathematik / Informatik
Studiengang Informatik

1. Gutachter: Prof. Dr. Rainer Koschke
2. Gutachter: Dr. Berthold Hoffmann

Erklärung

Ich versichere, die Diplomarbeit ohne fremde Hilfe angefertigt zu haben. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, sind als solche kenntlich gemacht.

Bremen, den 11.12.2006

. . . . . .
(Jan Harder)

INHALTSVERZEICHNIS

KAPITEL 1

Einleitung

Visual Basic ist eine populäre Sprache zum Erstellen von Windowsapplikationen. Nach Angaben des Herstellers Microsoft wurde allein die fünfte Version bis Anfang 1998 mehr als eine Million Mal lizenziert.[1] Dementsprechend groß dürfte auch die Menge von Applikationen sein, die über die Jahre in dieser Sprache realisiert wurden.

Im Zuge der Einführung des *.NET-Frameworks*, einer neuen, einheitlichen und objektorientierten Plattform und Laufzeitumgebung von Microsoft, wurde Visual Basic jedoch so grundlegend verändert, um der neuen Windowswelt gerecht zu werden, dass die Abwärtskompatibilität zu den früheren Versionen verloren ging. Tiefgreifende Veränderungen an der Syntax, aber vor allem auch der Semantik der Sprache machen eine direkte Migration vorhandener Programme unmöglich. Aus der vornehmlich prozeduralen Programmiersprache, die lediglich über eine rudimentäre Objektorientierung verfügte, wurde eine strikt objektorientierte Sprache. Zwar lassen sich alte Programme teilweise automatisiert in die neue Syntax transformieren (siehe [Mic06d]), allerdings führt kein Weg daran vorbei, die Semantik vielerorts von Hand anzupassen.

Die Migration vorhandener Applikationen zur neuen Version (oder einer vollkommen anderen Sprache) lässt sich nur noch begrenzt aufschieben, denn gemäß Microsofts Lifecycle Policy endet die Phase der aktiven Unterstützung und damit auch die Versorgung mit Aktualisierungen und Sicherheitspatches für die einzige Entwicklungsumgebung im März 2008. Zwar werden bestehende Programme auch noch einige Zeit darüber hinaus lauffähig bleiben, langfristig wird sich ein Umstieg aber nicht vermeiden lassen (siehe [Mic06e, Mic06c]).

Problematisch ist eine Migration aber nicht nur aufgrund des hohen Arbeitsaufwands, der damit verbunden ist. Oft sind Softwaresysteme zu groß, um für eine Person in ihrer Gänze überschaubar und verständlich zu sein. Zudem wird es notwendig sein auch solche Programmteile anzupassen, die über lange Zeit unverändert geblieben sind und über deren Funktionsweise eventuell gar kein vollständiges Wissen vorhanden ist – sei es weil die Personen mit diesem Wissen das Projekt verlassen haben, Dokumentation verloren ging oder vielleicht sogar nie in ausreichendem Maße erstellt wurde. Änderungen an Softwaresystemen und auch die Migration erfordern aber ein umfassendes Programmverständnis, das gegebenenfalls neu erworben werden muss – ein Prozess, der schon bei den üblichen Aufgaben der Softwarewartung, also im Wesentlichen dem Beheben von Fehlern und Hinzufügen neuer Funktionalität, etwa die Hälfte des Zeitaufwands ausmacht (vgl. [RVP06]).

[1] Siehe Pressemitteilung unter: `http://www.microsoft.com/presspass/press/1998/jan98/vb5mompr.mspx`

1.1 Das Bauhaus-Projekt

Gerade mit diesen Problemen beschäftigt sich das Bauhaus-Projekt (beschrieben in [RVP06, EKP+99]) der Universitäten Stuttgart und Bremen, in dessen Rahmen diese Diplomarbeit entstanden ist. Es entwickelt Werkzeuge und Methoden, die Wartungsingenieure bei diesen Aufgaben unterstützen sollen. Ein wesentliches Ziel besteht dabei darin, das Verstehen von Programmen durch geeignete Darstellungsformen und Analysen zu erleichtern und die gezielte Betrachtung und Überwachung bestimmter Programmmerkmale zu ermöglichen. Hierbei steht vor allem die Rückgewinnung der Systemarchitektur bestehender Softwaresysteme im Mittelpunkt.

Dem Wartungsingenieur stehen dazu automatische Analysen zur Verfügung, die darauf abzielen die Zusammengehörigkeit und die Beziehungen von Programm-Artefakten anhand verschiedener Kriterien zu ermitteln. Sie können dazu verwendet werden, bestimmte Muster und Merkmale in großen und komplexen Softwaresystemen zu erkennen und liefern letztendlich Vorschläge für mögliche architektonische Komponenten des Systems. So erhält der Wartungsingenieur ein architektonisches Grundmodell, das er manuell erweitern, korrigieren oder auch verwerfen kann, um schrittweise die Architektur des Softwaresystems zu rekonstruieren.

Genauso kann er aber auch eine erwartete oder erwünschte Architektur spezifizieren und sie durch Analysen mit der tatsächlichen Systemarchitektur abgleichen. Auf diese Weise lässt sich unter anderem die Einhaltung einer Soll-Architektur im Laufe eines Softwareprojekts überwachen. Zudem sollen allgemeine Wartungsaufgaben unterstützt werden, beispielsweise indem die globalen Auswirkungen von Änderungen an den Programmen durch Analysen abgeschätzt werden. Ebenso lassen sich qualitative und quantitative Merkmale des Quelltextes untersuchen und überwachen. Dies geschieht mit Hilfe von Softwaremetriken oder durch die Erkennung von dupliziertem Code – so genannten Klonen – in den Quelltexten.

Um die Analysen möglichst unabhängig von konkreten Programmiersprachen durchführen zu können, werden im Bauhaus-Projekt zwei eigene Repräsentationsformen für Programme eingesetzt, die Quelltexte auf unterschiedlichen Abstraktionsstufen darstellen. Hierbei bildet die ***I**nter**M**ediate **L**anguage* (IML) die Programme in sehr quelltextnaher Form ab, indem sie unter anderem den Kontroll- und Datenfluss nachbildet. Sie dient vornehmlich dazu, einheitliche Analysen auf einer technisch sehr detaillierten Ebene durchzuführen.

Die zweite Repräsentationsform ist der ***R**esource **F**low **G**raph* (RFG), der ausschließlich die globalen Bestandteile von Programmen, deren Attribute und ihre gegenseitigen Beziehungen abbildet, also eine wesentlich höhere Abstraktionsstufe wählt. Er kann mit dem Tool *Gravis* visualisiert werden und dient dazu, Programme sowie die Ergebnisse der Analysen darzustellen. Ein wesentliches Hilfsmittel zur übersichtlichen und verständlichen Darstellung sind dabei Sichten, die es ermöglichen, Teilgraphen des RFG darzustellen und diese als Ausgangspunkt für Analysen zu nutzen.

Die Erstellung des RFG für ein Programm ist auf unterschiedliche Weise möglich. Ein Weg besteht darin, die IML aus den Quelltexten zu generieren und aus ihr wiederum den RFG abzuleiten. Aber auch der direkte Weg vom Quelltext hin zum RFG ist möglich. Für die Sprache Java kann der Graph zudem auf Basis der Bytecode-Darstellung von Programmen generiert werden. Zum Zeitpunkt dieser Arbeit verfügt die Bauhaus-Suite über Frontends, die C, C++, Java und Ada in die projekteigenen Formate überführen können und es somit ermöglichen, Programme dieser Sprachen Analysen zu unterziehen

1.2 Aufgabenstellung

Die Zielsetzung dieser Arbeit ist es, die Generierung des Resource Flow Graph auch für Visual Basic 6 Programme zu ermöglichen. Diese spezielle Version der Sprache ist diejenige, die als letzte vor dem Bruch der Abwärtskompatibilität veröffentlicht wurde. Für ihre Programme besteht zukünftig, wie eingangs beschrieben, ein besonderer Migrationsbedarf. Die Aufgaben, die sich daraus ergeben können mit den Bauhaus-Werkzeugen unterstützt werden. Schon durch die alleinige Generierung von RFGs für Visual Basic 6 Programme wird es ermöglicht, viele der generischen Analysen auch für diese Sprache einzusetzen.

Ein großes Hindernis bei der Analyse von Visual Basic 6 Programmen besteht jedoch darin, dass die Sprache nur unvollständig dokumentiert ist. Problematisch ist dabei vor allem das Fehlen einer formalen Grammatik, die die Syntax beschreibt. Zugleich zeichnet sich die Sprache durch eine große Vielfalt an Sonderformen und Ausnahmen aus, die oft falsch oder gar nicht dokumentiert sind. Daher besteht der Schwerpunkt dieser Arbeit darin, die Analyse der Programme trotz dieser Ausgangslage überhaupt zu ermöglichen, indem entsprechende Tools geschaffen werden. Wünschenswert ist dabei, dass diese mit Rücksicht auf spätere Weiterentwicklungen konzipiert werden. Beispielsweise wäre langfristig auch die Generierung der IML denkbar, die wesentlich detailliertere Untersuchungen der Quelltexte erfordert.

Das Ergebnis der in dieser Arbeit zu erstellenden Analysewerkzeuge soll der RFG der untersuchten Programme sein. Als Voraussetzung hierfür muss ein Schema definiert werden, das Visual Basic 6 auf das RFG-Modell abbildet. Neben der Generierung der RFGs sollen zudem Softwaremetriken über die untersuchten Programme erhoben und ihre Ergebnisse an den Graphen annotiert werden. Konkret handelt es sich dabei um *Lines of Code*, die *zyklomatische Komplexität* (auch *McCabe-Metrik* genannt) und die *Halstead-Metrik*.

Beispielprogramme

Es ist wünschenswert, beliebige Visual Basic 6 Programme in ihre RFG-Repräsentation überführen zu können. Da es aufgrund der fehlenden Grammatik jedoch schwierig ist ein syntaktisch und semantisch vollständig und korrektes Analysewerkzeug zu entwickeln, wird eine feste Menge an Beispielprogrammen vorgegeben, deren Analyse als Ziel gilt. Hierbei handelt es sich um ein ein kommerzielles *Enterprise Resource Planning*-System[2] (kurz ERP-System) mit einem Umfang von 268.000 Zeilen Programmcode sowie die Visual Basic 6 Beispiele der *Microsoft Developer Network Library für Visual Studio 6.0*, die nochmals 40.000 Zeilen Programmcode beinhalten. Während das erste Projekt eine große Masse an Code darstellt, der jedoch recht homogen ist, decken die Microsoft-Beispiele die Breite der Sprachelemente und Einsatzmöglichkeiten ab. Zusammen sollten diese Quellen daher einen recht großen Teil der möglichen VB6-Syntax abdecken.

Vorgegebene Rahmenbedingungen

Durch die bereits vorhandenen Tools und Bibliotheken des Bauhaus-Projekts ergeben sich einige Vorgaben für die Umsetzung dieser Arbeit. So soll die Implementierung des Systems in den Sprachen C++ und Ada stattfinden. Die Verwendung von Ada ist insbesondere dort erforderlich, wo auf bestehende Bauhaus-Bibliotheken zugegriffen werden muss. Die Übersetzung der Ada-Sourcen ist aufgrund der eingeschränkten Verfügbarkeit des Ada-Compilers derzeit

[2] Unter dem Begriff der *Enterprise Resource Planning*-Software versteht man solche Systeme, die dazu dienen alle Daten und Prozesse einer Organisation in einer Software zu vereinen.

nur in einer Linux-Umgebung möglich. Daher muss die Entwicklung auf einem Linux-System stattfinden.

1.2.1 Konkrete Ziele

Aus der Aufgabenstellung ergeben sich im einzelnen die folgenden Ziele:

- **Abbildung von Visual Basic 6 auf den RFG**

  Bevor der RFG für Visual Basic 6 Programme generiert werden kann, muss festgelegt werden, wie die Programme und ihre einzelnen Bestandteile auf das RFG-Modell abgebildet werden können. Dies muss in Form eines RFG-Schemas geschehen, wie es bereits für andere Sprachen existiert. Diese vorhandenen Schemata sollen dabei als Richtlinie für die Abbildung gelten.

  Sollten sich Visual Basic 6 Programme nicht mit den bereits vorhandenen Konzepten des RFG-Modells abbilden lassen, so muss dieses entsprechend erweitert werden.

- **Analyse von Visual Basic 6 Programmen**

  Um die für den RFG benötigten Daten erheben zu können, muss eine Analyse von Visual Basic 6 Programmen durchgeführt werden. Bislang fehlen die hierzu notwendigen Grundlagen, beispielsweise in Form einer ausreichend genauen Grammatikdefinition. Zudem ist die Dokumentation lücken- und fehlerhaft, was insbesondere die Untersuchung semantischer Aspekte verhindert. Diese fehlenden Grundlagen müssen im Rahmen dieser Arbeit erarbeitet werden. Auf dieser Basis sind dann Analysewerkzeuge zu implementieren, die alle für den RFG benötigten Daten aus den Programmen extrahieren. Hierbei sollen ausschließlich statische Analysemethoden zum Einsatz kommen. Die Untersuchung von laufzeitabhängigen Aspekten gehört nicht zu den Aufgaben dieser Arbeit.

  Die Implementierung soll in der Sprache C++ erfolgen. Es ist zudem wünschenswert, wenn auch nicht zwingend erforderlich, dass die Analysewerkzeuge später auch für komplexere Untersuchungen erweitert werden können.

  Im Rahmen dieser Arbeit sollen die zuvor definierten Testprogramme mit diesen Analysewerkzeugen untersucht werden können.

- **Erhebung von Metriken**

  Die drei Metriken Lines of Code, zyklomatische Komplexität und die Halstead müssen über die Prozeduren der Programme erhoben werden.

- **Generierung des RFG für Visual Basic 6 Programme**

  Es muss ein Werkzeug implementiert werden, das auf Grundlage der Ergebnisse der Analyse von Visual Basic 6 Programmen die RFG-Repräsentation generiert. Die erhobenen Metriken sind ebenfalls an diesen Graphen zu annotieren.

1.3 Aufbau der Arbeit

Diese Arbeit teilt sich neben dieser Einleitung in fünf weitere Kapitel auf. Das folgende Kapitel 2 beschreibt die wesentlichen Grundlagen. Zunächst wird hier eine Einführung in Visual Basic

6 gegeben, wobei auch die Ausgangslage für die Analyse, sowie für diese Arbeit relevante Probleme, die sich aus der Sprache ergeben, näher betrachtet werden. Ebenso werden einige Aspekte des Component Object Models beleuchtet, das ebenfalls für die Analyse von Visual Basic 6 Programmen von Bedeutung ist. Im weiteren Verlauf wird der Resource Flow Graph vorgestellt und ein kurzer Ausblick auf ähnliche Repräsentationsformen gegeben. Abschließend stelle ich in diesem Grundlagenkapitel die Metriken, deren Erhebung in der Aufgabenstellung gefordert wird, einzeln vor.

Das dritte Kapitel widmet sich der Erstellung eines Schemas. dass Visual Basic 6 auf den RFG abbildet. Dazu stelle ich das existierende Schema für die Sprache C++ vor und leite daraus ein neues für Visual Basic her. Die Abbildung der einzelnen Programmartefakte auf die RFG-Konzepte wird hierbei im Detail definiert.

Kapitel 4 behandelt mit der Analyse von Visual Basic 6 den Schwerpunkt dieser Arbeit. Hier betrachte ich zunächst Vorgehensweisen zum Umgang mit unvollständigem Wissen bei der Programmanalyse beziehungsweise im Reverse Engineering allgemein. Aus diesen Vorgehensweisen leite ich dann eine Methodik ab, die der Ausgangssituation in dieser Arbeit angepasst ist und beschreibe diese. Das Ergebnis – eine neue Grammatik für Visual Basic 6 – stelle ich vor, indem ich Beispiele für einzelne Problemlösungen erläutere.

Der zweite Teil dieses Kapitels widmet sich der semantischen Analyse der Programme, die alle für den RFG benötigten Daten sammelt und herleitet. Hier liegt der Schwerpunkt auf der Untersuchung der im Programm verwendeten Symbole und ihrer Beziehungen, die eine Auflösung von Bezeichnern im Quelltext erfordert. Auch hier steht nur kein vollständiges Wissen – in diesem Fall über die Semantik der Sprache – zur Verfügung. Daher beschreibe ich auch in diesem Teil mit welchen Methoden vorgegangen bin, um eine semantische Analyse durchführen zu können und stellen diese dann wiederum exemplarisch vor. Die Erhebung der Metriken wird im Anschluß daran erläutert.

In Kapitel 5 befasse ich mich mit der Generierung des RFG aufgrund der Daten, die bei der VB6-Analyse extrahiert wurden. Das sechste und letzte Kapitel betrachtet schließlich die Ergebnisse dieser Arbeit und bewertet sie. Zudem gebe ich dort einen Ausblick darauf welche Möglichkeiten sich in Folge dieser Arbeit eröffnen.

Der Anhang dieser Arbeit enthält, neben den obligatorischen Verzeichnissen, auf Lesbarkeit optimierte Fassungen der erarbeiteten Grammatiken.

1.3.1 Begriffe

Mit Visual Basic bezieht sich diese Arbeit im Kern auf ein Produkt der Firma Microsoft und sein technisches Umfeld. Daher kommen im Text häufig Produktnamen zur Verwendung. Durch regelmäßige Umbenennungen und eine uneinheitliche Namensvergabe sind manche dieser Bezeichnungen irreführend. Deshalb möchte ich gleich zu Beginn einige Konventionen für ihre Verwendung festlegen, um missverständliche Formulierungen zu vermeiden. Es handelt sich dabei um die folgenden Begriffe:

- **Visual Basic**

  Der Begriff *Visual Basic*, ohne den Zusatz einer konkreten Versionsbezeichnung, wird außerhalb dieser Arbeit sehr unterschiedlich verwendet. Teilweise als Oberbegriff für alle Versionen, anderenorts als Sammelbegriff für alle Versionen bis einschließlich Sechs verwendet und an wieder anderen Stellen bezeichnet er die zum Zeitpunkt der Textverfassung aktuelle Version. Um Verwirrung zu vermeiden, gelten in dieser Arbeit die folgenden Konventionen:

- *Visual Basic* bezeichnet, soweit nicht anders erwähnt, Version sechs, deren Analyse Gegenstand dieser Arbeit ist.
- *VB6* ist Kurzform für Visual Basic 6.
- *Visual Basic .NET* bezeichnet die Gruppe aller Nachfolgerversionen von Visual Basic 6, die auf dem .NET-Framework basieren und nach dem erwähnten Kompatibilitätsbruch erschienen sind. In dieser Arbeit werden sie nur zusammengefasst als Gruppe betrachtet. Diese Versionen sind zum Zeitpunkt dieser Arbeit: *Visual Basic .NET*, *Visual Basic .NET 2003* und *Visual Basic 2005*.
- *VB.NET* ist Kurzform für Visual Basic .NET, in der obigen Bedeutung.

Alle sonstigen Versionen werde ich stets mit ihrer vollen Versionsbezeichnung benennen.

- **COM, OLE, ActiveX**

  Die Bedeutung dieser Begriffe hat sich laut [Kof00, S. 872ff] und [Mon01, S. 90ff] immer wieder geändert. *ActiveX* ist demnach mehr als breit eingesetzte Marketingbezeichnung zu verstehen, denn als Bezeichnung zusammengehöriger Konzepte. In den meisten Fällen wird es jedoch für Konzepte verwendet, die eng mit dem *Component Object Model* (COM) oder dem *Object Linking and Embedding* (OLE) verbunden sind. Alle drei Begriffe werden zumeist für die gleichen Konzepte verwendet.

  Die Bedeutung der drei Begriffe ist mitunter vom Zeitpunkt ihrer Verwendung abhängig. Dies bedeutet für diese Arbeit, die acht Jahre nach Erscheinen der untersuchten Visual Basic Version entstanden ist und unterschiedlich alte Quellen verwendet, eine Erschwernis. Ich habe mich daher entschieden, die Bezüge zu COM, OLE und ActiveX, die in der verwendeten Literatur zu finden sind, auf COM als einheitlichen Begriff abzubilden, um die Namenswirren nicht auch in diese Arbeit zu übertragen. Dabei ist prinzipiell der Standard des Component Object Models gemeint, der in Visual Basic 6 zur Verwendung kommt, um Bibliotheken einzubinden.

1.3.2 Genus

Zur besseren Lesbarkeit und somit auch zugunsten einer besseren Verständlichkeit des Textes wird in dieser Arbeit das generische Maskulinum bei der Bezeichnung von Personen oder Personengruppen verwendet, sofern diese nicht mit dem Neutrum zu bezeichnen sind. Dies hat keinen wertenden Hintergrund, so dass Frauen stets ebenfalls in die Aussage einbezogen sind.

KAPITEL 2

Grundlagen

In diesem Kapitel beschreibe ich die Grundlagen, die für diese Arbeit benötigt werden. Das Ziel dabei ist es vor allem einen Überblick über die zu untersuchende Sprache und das Modell, auf das diese abgebildet werden soll, zu vermitteln, da diese der Arbeitsgegenstand sind. Der Schwerpunkt liegt dabei jeweils auf den Aspekten, die für diese Arbeit von Bedeutung sind.

Zunächst werde ich mit Visual Basic 6 den Untersuchungsgegenstand der zu erstellenden Analysewerkzeuge vorstellen. Dabei beschreibe ich neben der Sprache im Allgemeinen auch die Besonderheiten, die für eine Abbildung auf den RFG und die Analyse der Sprache von wesentlicher Bedeutung sind und deren Kenntnis für das Verständnis dieser Arbeit notwendig ist. Zudem gehe ich auf die bestehenden Voraussetzungen für die Analyse der Sprache ein. Durch das eng mit Visual Basic 6 verbundene Component Object Model ergeben sich ebenfalls einige Probleme und Anforderungen, derer ich mich in dieser Arbeit annehmen will. Diese werden im zweiten Abschnitt dieses Kapitels aufgezeigt. Eine weitere wichtige Grundlage ist das RFG-Metamodell, das ich in Abschnitt 2.3 beschreibe. Hierbei soll auch ein vergleichender Blick auf ähnliche Modelle geworfen werden, die außerhalb des Bauhaus-Projektes entworfen wurden. Schließlich werde ich die Metriken, die über die Visual Basic Programme erhoben werden sollen, kurz vorstellen.

Kapitelinhalt

2.1 Visual Basic 6

Visual Basic 6 ist eine proprietäre Programmiersprache von Microsoft, die zur Programmierung von Windowsapplikationen mit grafischer Benutzeroberfläche dient. Schon der Name macht deutlich, dass sie als Fortentwicklung von *BASIC*, dem *Beginners all purpose symbolic instruction code* - einer populären, imperativen Programmiersprache aus den 1960er Jahren, die sich vor allem an Anfänger wendete - zu verstehen ist. Visual Basic versucht die Einfachheit dieses Vorbildes zu erhalten, gleichzeitig aber Konzepte höherer Programmiersprachen zu bieten (vgl. [Dud01]). Dies zeigt sich durch wesentliche Erweiterungen wie einem strukturierten und eventgesteuerten Programmaufbau, die Möglichkeit graphische Benutzeroberflächen zu erstellen und sogar durch Ansätze zur objektorientierten Programmierung.

Die Bezeichnung „visual“ bezieht sich dabei auf die fest mit der Sprache verbundene Entwicklungsumgebung *Visual Studio*, die eine Einheit mit dem einzig verfügbaren Compiler für Visual Basic 6 bildet. Sie bietet mit dem so genannten *Designer* ein Entwicklungswerkzeug, das es erlaubt Programmfenster in einer visuellen Darstellung aus Komponenten nach dem Drag&Drop-Prinzip zusammenzusetzen und deren Attribute in Eigenschaftsfenstern festzulegen. Dieser Teil der Implementierung muss vom Programmierer also nicht mehr anhand des Quelltextes vorgenommen werden. Zudem werden Standardaufgaben durch Assistenten vereinfacht, indem diese Quelltextvorlagen erstellen.

Nach der Klassifikation von visuellen Programmiersystemen in [Sch98, S. 23ff], ist Visual Basic keine *visuelle Programmiersprache* im eigentlichen Sinn. Dieser Begriff wird in der Wissenschaft üblicher Weise für Sprachen verwendet, die den gesamten Programmierprozess, inklusive der Erstellung der Logik, auf visueller Basis umsetzen, ohne dabei nach außen auf eine textuelle Programmrepräsentation zurückzugreifen. Die Sprache Visual Basic und die Entwicklungsumgebung sind demnach zu den *visuellen Programmiersystemen mit verbaler (textueller) Programmiersprache* zu zählen.

Visual Basic hat einige Ableger, die für spezielle Einsatzgebiete konzipiert wurden. So gibt es beispielsweise ein *Visual Basic for Applications*, mit dem sich Macros in Microsofts Office-Programmen definieren lassen. *Viusal Basic Script* stellt eine Scriptsprache dar, die in Webseiten eingebettet werden kann. Diese Varianten unterscheiden sich jedoch von Visual Basic 6, dessen Schwerpunkt vollwertige Windows-Anwendungen sind. In dieser Arbeit befasse ich mich ausschließlich mit letzterer Version.

Dieser Abschnitt soll einen allgemeinen Überblick über die Sprache bieten und darüber hinaus alle Aspekte vorstellen, die für diese Arbeit relevant sind. Dazu gehört die Vorstellung des Programmaufbaus und einiger Besonderheiten, die für die Analyse von Visual Basic Programmen und die Generierung ihrer RFGs von Bedeutung sind. Weiterhin werde ich das Problem der unzureichenden Dokumentation näher beschreiben und dabei die verfügbaren Grammatiken für die Sprache vorstellen und bewerten. Die Informationen in diesem Abschnitt sind, soweit nicht anders gekennzeichnet, aus den Quellen [Mic06b, Mon01, Mas99, Kof00, Cow00, Hau99] zusammengetragen.

2.1.1 Syntaxbeispiel

Um Visual Basic ein wenig transparenter zu machen, beginne ich die Beschreibung mit einem kurzen Syntaxbeispiel, das einen ersten Eindruck vermittelt und dabei hilft, die im weiteren Verlauf folgenden Beispiele besser zu verstehen. Listing 2.1 zeigt eine Funktion, die die Fakultät einer Zahl berechnet und als Ergebnis zurückliefert.

```
Public Function Fakult(zahl As Integer) as Integer

    Dim tmp As Integer
    Dim i   As Integer

    tmp = 1

    For i = 1 To zahl
        If i = 9 Then
            Fakult = -1    '9! > 2^16 = max. Integer-Wert
            Exit Function 'gebe -1 zrueck, wenn zahl zu gross
        End If
        tmp = tmp * i
    Next

    Fakult = tmp           'setze Rueckgabewert

End Function
```

Listing 2.1: Berechnung der Fakultät einer Zahl in VB6

Die Syntax ist schlüsselwortbasiert und verwendet Zeilenumbrüche zur Beendigung von einzelnen Ausdrücken. Somit ergibt sich ein Bild, das dem ursprünglichen BASIC noch immer in gewisser Weise ähnelt. In meinen Beispielen werde ich stellenweise Anmerkungen und Hinweise in Form von Kommentaren einstreuen. Diese werden immer von einem Hochkomma angeführt. Außerdem ist anzumerken, dass ich in vielen Beispielen die `Dim`-Anweisung verwende, die Variablen deklariert. Ebenso sei erwähnt, dass der Rückgabewert von Funktionen definiert wird, indem er dem Namen der Funktion zugewiesen wird.

2.1.2 Programmaufbau

Visual Basic 6 lässt sich nicht auf einen einzigen Sprachtyp festlegen. Es ermöglicht eine prozedurale Programmierung, einige Teile folgen jedoch objektorientierten Prinzipien. Die Möglichkeiten der objektorientierten Programmierung sind allerdings sehr begrenzt und bieten beispielsweise keine echte Vererbung und nur eine sehr begrenzte Polymorphie.

2.1.2.1 Prozedurale Programmierung

Der prozedurale Anteil von Visual Basic 6 wird in so genannte *Standardmodule* untergliedert. Diese stellen einfache Sammlungen von Variablen, Konstanten, Prozeduren und Typen dar. Diese Bestandteile können jeweils global im gesamten Programm verfügbar gemacht werden, also auch in allen anderen Modulen oder nur modulintern Geltung besitzen. Dabei ist die globale Verfügbarkeit der Standard. Der ausführbare Programmanteil ist grundsätzlich in Prozeduren unterteilt. *Prozedur* wird in der Dokumentation dabei als Oberbegriff für alle Ausprägungen von Unterprogrammen und Funktionen verwendet. Die beiden gängigsten Arten sind die `Sub`-Prozedur – ein reines Unterprogramm mit Parametern aber ohne Rückgabewert – sowie die `Function`-Prozedur, die zusätzlich über einen Rückgabewert verfügt.

2.1.2.2 Objektorientierte Programmierung

Neben den Standardmodulen kennt Visual Basic auch Klassenmodule, deren Inhalt jeweils eine Klasse definiert. Eine besondere Form von Klassen stellen die Formularmodule dar, die dazu verwendet werden Formulare – konkret die Fenster der Applikation – zu definieren. Hierbei handelt es sich letztendlich ebenfalls um Klassen, die jedoch über einen speziellen Bereich im Kopf des Modulquelltextes verfügen, in dem sich die Deklarationen für die visuell erstellten Oberflächenkomponenten befinden. Dieser Kopfteil des Quelltextes wird vollautomatisch gepflegt und dem Programmierer in Visual Studio nicht angezeigt. Er wird nur in anderen Texteditoren sichtbar.

Syntax und Anweisungen sind mit denen der Standardmodule identisch, lediglich die Semantik ändert sich durch die Instanzgebundenheit. Neben den bereits genannten Prozedurarten können in Klassen zusätzlich `Property`- und `Event`-Prozeduren verwendet werden[1].

Propertys

Eine *Property*, also eine Eigenschaft, ist ein Konstrukt, das sich von außen betrachtet wie eine normale Variable oder ein Klassenattribut verhält. Tatsächlich wird es jedoch durch die Implementierung von Zugriffsmethoden implizit deklariert. Pro Zugriffsart – in Visual Basic 6 sind dies `Get` (Zugriff), `Let` (Wertzuweisung) und `Set` (Setzen einer Objektreferenz) – lässt sich eine Zugriffsprozedur definieren. Mehrere solche Prozeduren, die den gleichen Namen tragen, definieren implizit die Property. Es ist dabei unerheblich, was darin tatsächlich geschieht. Dies bleibt dem Programmierer überlassen. Somit gibt es auch keine Daten, die einer Property automatisch zugeordnet sind. Die zum Speichern der Eigenschaftsdaten benötigten Datenstrukturen sind vom Programmierer zu schaffen. Visual Basics Property-Konzept ist daher mit der üblichen Praxis der objektorientierten Programmierung zu vergleichen, bei dem Accessormethoden für Klassenattribute definiert werden, um das Geheimnisprinzip zu wahren. Listing 2.2 zeigt ein Beispiel für die Deklaration und Verwendung einer Property.

```
Private mvarAngle As Float

Public Property Get angle () As Float
    angle = mvarAngle
End Property

Public Property Let angle (f As Float)
    'only accept values between 0 and 360
    If f < 0 Or f > 360 Then
        'raise an error...
    Else
        mvarAngle = f
    End If
End Property

Public Sub increaseAngle (amount As Float)
    angle = angle + amount 'left use of angle implicity calls Let, right use Get
End Sub
```

Listing 2.2: Propertys in VB6

[1] Tatsächlich können Propertys in Visual Basic 6 auch in Standardmodulen verwendet werden, ihr Haupteinsatzbereich sind aber Klassen

Events

Ereignisse spielen in Visual Basic 6 eine zentrale Rolle, da sie das wichtigste Mittel zur Steuerung des Programmablaufs von Anwendungen mit grafischer Benutzeroberfläche darstellen. Diese Form der Programmsteuerung benötigt drei Elemente: Eine Klasse oder ein Formular, das das *Event* (gelegentlich auch *Eventprozedur* genannt) deklariert, eine Instanz eben dieser Klasse bzw. des Formulars und ein weiteres Modul, das diese Instanz erzeugt und eine `Sub`-Prozedur zur Behandlung des Events exklusiv für diese Instanz implementiert. Diese Zusammenhänge werden im Folgenden näher erläutert.

Die Deklaration eines Events beschränkt sich auf die Angabe einer Signatur mit dessen Namen und Parametern (siehe Listing 2.3). Beliebige Module können dann Variable vom Typ der Klasse oder des Formulars deklarieren und dabei explizit angeben, dass sie über mögliche Events informiert werden möchten. Die Behandlung von Events erfolgt, indem eine `Sub`-Prozedur mit einem bestimmten Namensmuster, das aus dem Namen der Variable und dem Namen des zu behandelnden Events besteht, definiert wird. Listing 2.4 zeigt eine Prozedur, die das `Click`-Event für das von der Variable `Button1` referenzierte Objekt behandelt. Eine solche Prozedur behandelt also stets ein bestimmtes Event für eine bestimmte Variable. Sobald das Event innerhalb seiner Klasse durch das `RaiseEvent`-Statement ausgelöst wird, erfolgt ein Aufruf dieser Prozedur(siehe Listing 2.3).

```
Public Event Click()
...

Private Sub beingClicked ()
   ...
   RaiseEvent Click
   ...
End Sub
```

Listing 2.3: Deklaration und Auslösung eines Events in einer VB6-Klasse oder einem Formular

```
Private WithEvents Button1 As New Control 'declare an event aware Control instance

...

Sub Button1_Click()
    'handle the click event of Button1 here...
End Sub
```

Listing 2.4: Eventbehandlung in VB6

Visual Basic 6 Anwendungen warten nach ihrer Initialisierung auf die Auslösung von Events. Über sie lassen sich, wie in den vorigen Listings schon zu sehen war, Ereignisse an der Benutzeroberfläche erkennen und behandeln. Die dabei auftretenden Konsequenzen, die von der eventbehandelnden `Sub`-Prozedur angestoßen werden, können beliebig komplex sein.

Vererbung

Visual Basic 6 kennt keine Vererbung von Daten und Funktionalität, dennoch unterstützt es Polymorphie. Über die `Implements`-Anweisung kann eine beliebige Anzahl von Klassen angegeben werden, deren Interfaces implementiert werden. Die Implementierung wird vorgenommen, indem den Methoden der Name ihrer Ursprungsklasse vorangestellt wird. Angenommen die Klasse `Cat` stellt eine Methode `Speak` zur Verfügung und die Klasse `Lion` soll von `Cat` abgeleitet werden. So muss `Lion` wie in Listing 2.5 dargestellt aussehen.

```
Implements Cat

Private Sub Cat_Speak()
    MsgBox "ROAR!"
End Sub
```

Listing 2.5: Implementierung von Klasseninterfaces in VB6

Die Methode `Cat_Speak` definiert das Verhalten, das auftritt, wenn die `Lion`-Instanz über das `Cat`-Interface angesprochen wird. Die Implementierung ist allein dem Programmierer überlassen. Selbst wenn das Verhalten nicht von dem in `Cat` definierten abweichen soll, so muss es entsprechend implementiert werden. Hierzu kann der Code aus `Cat` kopiert werden oder in `Lion` eine Instanz der Klasse `Cat` gehalten und aufgerufen werden. Durch die Existenz von `Cat_Speak` erhält `Lion` jedoch keine eigene `Speak`-Methode, die sich über das `Lion`-Interface aufrufen ließe. Um eine vollständige Vererbung zu simulieren, muss der Programmierer selbst eine `Speak`-Methode implementieren, die `Cat_Speak` aufruft. Es findet also keine Vererbung statt, allerdings erlaubt diese Konstruktion Polymorphie. Wird die Methode `Speak` an einer Objektvariable vom Typ `Cat` aufgerufen, so muss zur Laufzeit ermittelt werden, welchen Typ das referenzierte Objekt tatsächlich hat, um die aufzurufende `Speak`-Implementierung zu ermitteln.

2.1.3 Besonderheiten

Visual Basic 6 ist voll von Besonderheiten, die oft erst bei einer genaueren Betrachtung auffallen und selten dokumentiert sind. Dies beginnt bei der Vielfalt an Anweisungen und Abkürzungsformen, die sich überall in der Syntax finden. Als einfaches Beispiel ist es möglich, die Deklaration `Dim i As Integer` mit `Dim i%` abzukürzen. Für die Definition einer While-Schleife bieten sich gleich mehrere syntaktische Formen an und an gewissen Stellen können bestimmte Prozeduren ohne die Angabe ihres Namens aufgerufen werden. Hinzu kommen semantische Besonderheiten, die mitunter recht ungewöhnlich erscheinen. So lässt sich beispielsweise der Typ von Variablen über den Anfangsbuchstaben ihres Bezeichners steuern oder die Zahl der ersten Array-Index modulweit umdefinieren.

Die Beschreibung aller Sonderfälle und Eigenarten, die im Rahmen dieser Arbeit berücksichtigt werden mussten, würde den Rahmen dieses Dokuments sprengen. Stattdessen möchte ich an dieser Stelle eine Auswahl von Besonderheiten anführen, auf die ich im weiteren Verlauf der Arbeit zurückkommen werde. Sei es weil sie besondere Auswirkungen auf die Erstellung des RFG-Schemas haben oder weil sie erwähnenswerte Schwierigkeiten bei den Analysen oder der Grammatikerstellung hervorrufen und daher später im Text als Beispiele aufgegriffen werden. Zudem sollen sie dabei helfen, die Vielfalt und Unordnung in Visual Basic 6 zu illustrieren.

Uneinheitliche und mehrdeutige Syntax

An manchen Stellen ist die Syntax von semantisch ähnlichen Sprachelementen uneinheitlich, so unterscheidet sich etwa der Aufruf von `Sub`-Prozeduren, die reine Unterprogramme ohne Rückgabewert darstellen, und `Function`-Prozeduren durch die Klammerung ihrer Argumente. Nur Funktionsargumente werden in Klammern gesetzt, die von `Sub`-Prozeduren dagegen nicht. Allerdings gibt es Ausnahmen: Stellt man einem `Sub`-Aufruf die `Call`-Anweisung voran, so ändert sich an der Semantik nichts, allerdings müssen die Argumente nun mit Klammern umschlossen werden. Hat eine `Sub` oder `Function` dagegen keine Parameter, so ist die Verwendung von einer leeren Klammerung optional. Diese Verwirrenden Regeln werden zusätzlich

noch durch eine semantische Besonderheit verkompliziert, durch die einzelne in Klammern gesetzte Argumente stets als Wert und nie als Referenz übergeben werden. Ob der Parameter eigentlich ein Referenzparameter ist, spielt dabei keine Rolle. Listing 2.6 zeigt einige Beispiele für diese uneinheitlichen Regeln.

```
Sub quadriere_sub (ByRef i as Integer)
    i = i * i
End Sub

Function quadriere_func (ByRef i as Integer) As Integer
    quadriere = i * i
End Sub

Sub eineAndereSub()

    Dim i As Integer

    '--- Syntaktisch korrekte Anweisungen: ---
    i = 2
    quadriere_sub i          ' i = 4
    Call quadriere_sub(i)    ' i = 16
    quadriere_sub(i)         ' call by value! i = 16
    Call quadriere_sub((i))  ' call by value! i = 16

    i = 2
    i = quadriere_func(i)    ' i = 4

    '--- Beispiele fuer syntaktisch falsche Anweisungen: ---
    SubMitZweiParametern(arg1, arg2)       ' Klammerung nicht erlaubt
    Call SubMitZweiParametern arg1, arg2   ' Klammerung erforderlich

    '--- Syntaktisch korrekte, aber nicht unterscheidbar: ---
    x = y ' y kann sein: Variable oder Function bzw. Property ohne Parameter
          ' x ist nur durch Kontext (links in einer Zweisung) als Variable zu erkennen
            .

End Sub
```

Listing 2.6: Ungleichmäßige Syntax in VB6

Insgesamt ist die Syntax recht uneinheitlich und erlaubt es nicht immer festzustellen, ob ein Bezeichner einen Prozeduraufruf oder eine Variablenverwendung repräsentiert. Das Durcheinander bei den Klammerungen führt insbesondere bei der syntaktischen Analyse zu einigen Komplikationen.

Bang-Operator und Standardattribute

Der Bang-Operator „`!`“– auch *Dictionary Lookup Operator* genannt (vgl. [Vic03]) – ist ein Beispiel für die Abkürzungsformen. Er soll den Zugriff auf die Elemente von *Aufzählungsklassen* – beispielsweise Collections – vereinfachen. Angenommen `fruits` stellt eine Collection dar, die ein Element mit dem Schlüssel `banana` enthält, so lässt sich dieses auf dem normalen Wege durch `fruits.Item("banana")` zugreifen. Äquivalent ist aber auch folgende Form mit dem Bang-Operator: `fruits!banana`. Ebenso erlaubt ist das äquivalente `fruits("banana")`.

Letztere Form stellt den Zugriff auf das Standardattribut dar. Jede Klasse kann eine ihrer Propertys zum Standard erheben, wodurch es ermöglicht wird, sie aufzurufen, indem die Argumentenliste für den Property-Aufruf direkt an die Instanzvariable angehängt wird. Gesetzt wird solch ein Standardattribut, indem der Objektinstanz ein Wert zugewiesen wird. Dies ist möglich, da die Zuweisung von Objektreferenzen über eine eigene Anweisung namens `Set` vorgenommen wird und nicht über die Standardzuweisung.

Um die Verwirrung um die Klammerungen perfekt zu machen sei angemerkt, dass auch beim

Zugriff auf Arrayfelder runde Klammern verwendet werden. Rein syntaktisch und ohne Berücksichtigung der Semantik betrachtet könnte ein Ausdruck `x(y)` also folgende Bedeutungen haben:

- Den Aufruf einer `Function` oder `Property` namens `x` mit Parameter `y`.
- Den Aufruf einer `Sub` namens `x` mit Parameter `y` als erzwungenen Wertparameter.
- Den Zugriff auf den Index `y` in einem Array namens `x`.
- Den Zugriff auf die Standardeigenschaft eines Objektes `x`

Die Kombination dieser Sonderformen kann zudem zu Ausdrücken wie beispielsweise `w(x)(y)(z)` führen. Hierbei könnte `w(x)` eine Funktion sein, die ein Objekt zückliefert, dessen Standardeigenschaft mit `(y)` aufgerufen wird, die wiederum ein Array zurückliefert, dessen Index `z` zugegriffen wird. Diese Besonderheiten müssen in der semantischen Analyse einzeln untersucht werden, da einige der Klammerungen Aufrufe repräsentieren, die für den RFG von Bedeutung sind.

Bezeichner und Schlüsselwörter

Ein vollkommen anderes Problem besteht darin, dass Schlüsselwörter nicht zwingend auch reserviert sind. Hier gilt offenbar kein festes Regelwerk, vielmehr ist es vom Kontext abhängig, wann welche Schlüsselwörter auch als Bezeichner verwendet werden dürfen. Die Regeln sind im Allgemeinen sehr liberal. Listing 4.6 zeigt ein Beispiel hierfür.

```
Sub Sub()                 'Compilerfehler
   'Anweisungen...
End Sub

Type myType
   Sub As String          'erlaubt
End Type

Sub Property()            'erlaubt
   'Anweisungen...
End Sub

Sub anotherSub()
   Property               'Compilerfehler
   Call Property          'erlaubt
End Sub
```

Listing 2.7: Kontextabhängige Reservierung von Schlüsselwörtern

Die genauen Regeln, denen die Reservierung von Schlüsselwörtern folgt sind nicht dokumentiert. Die Analyse von VB6-Programmen muss allerdings hiermit umgehen können, um Bezeichner korrekt zu erkennen. Auch die Erhebung bestmmter Metriken kann nur erfolgen, wenn eine Unterscheidung von Bezeichnern und Schlüsselwörtern erfolgt. Ich werde dieses Beispiel im Verlauf der Arbeit daher an mehreren Stellen wieder aufgreifen.

Implizite Variablendeklarartion

Auf lokaler Ebene können Variablen implizit durch ihre Verwendung deklariert werden. Dabei erhalten sie automatisch den generischen Datentyp `Variant`, der jeden Wert annehmen kann.

Objektinstanziierung

Zu jedem Formularmodul erstellt Visual Basic zur Laufzeit eine Standardinstanz, die den gleichen Namen trägt wie das Modul. So gibt es zu einem Formular `Form1` auch stets eine globale Variable gleichen Namens, die eine automatisch erzeugte Instanz des Formulars beinhaltet. Sofern nur eine einzelne Instanz eines Formulars benötigt wird, muss sich der Programmierer nicht mit deren Erstellung befassen.

Visuell erstellte Programmteile

Die Deklarationen und Initialisierungen der Formularkomponenten, die über die visuellen Werkzeuge der IDE erzeugt wurden, können wie schon angedeutet ebenfalls den Quelltexten entnommen werden. Sie werden mir einigen anderen Optionen und Attributen im Kopfbereich des Moduls abgelegt und von Visual Studio verwaltet. Der Programmierer bekommt sie jedoch nur zu Gesicht, wenn er die Module in einem anderen Editor betrachtet.

Die verwendete Syntax ist menschenlesbar, unterscheidet sich aber von der übrigen Syntax der Sprache. In einer Struktur aus Blöcken und Zuweisungen werden hier die einzelnen Komponenten deklariert und ihre Attribute mit konstanten Werten initialisiert. Jede Komponente definiert dabei einen eigenen Block, in dem ihre Attributzuweisungen enthalten sind. Außerdem können dort weitere Komponentendeklarationen folgen, wobei die Schachtelung die Zusammengehörigkeit definiert. So kann ein Rahmen beispielsweise einen Knopf enthalten. Allerdings sind alle definierten Komponenten ungeachtete ihrer Schachtelung Member des Formulars. Listing 2.8 zeigt ein einfaches Beispiel für einen Formularkopf.

```
VERSION 5.00
Begin VB.Form Form1
   Caption         =   "Form1"
   ClientHeight    =   2820
   ClientLeft      =   2550
   ClientTop       =   2730
   ClientWidth     =   2850
   LinkTopic       =   "Form1"
   ScaleHeight     =   188
   ScaleMode       =   3   'Pixel
   ScaleWidth      =   190
   Begin Threed.SSPanel pnCal
      Height          =   2400
      Left            =   120
      TabIndex        =   0
      Top             =   210
      Width           =   2655
      _Version        =   65536
      _ExtentX        =   4683
      _ExtentY        =   4233
      _StockProps     =   15
      Caption         =   "SSPanel1"
      BackColor       =   13160660
      Autosize        =   3
   End
End
Attribute VB_Name = "Form1"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = False
Option Explicit
```

Listing 2.8: Beispiel für einen Formularmodul-Kopf

Diesen besonderen Bestandteil von Formular-Quelltexten bezeichne ich in dieser Arbeit der

Einfachheit halber als *Designer-Code*, da sich das Werkzeug, mit dem er erstellt wird, „Designer “ nennt. Syntax und Sematik dieses Programmteils sind kaum dokumentiert. Dies ist ein Problem, weil Visual Basic hier, wie im Beispiel zu sehen ist, Attribute der Komponenten (`Caption`) mit eigenen Steuerbefehlen (`Top` oder `Version`) vermischt. Beides lässt sich nur unterscheiden, wenn alle Steuerbefehle bekannt sind.

Dies ist ein typisches Beispiel für die Probleme, die die Besonderheiten bei der Analyse aufwerfen. In diesem speziellen Fall war im ersten Moment gar nicht unbedingt ersichtlich, dass es sich hier zum Teil nicht um Attribute der verwendeten GUI-Komponente handelt. Solche Besonderheiten fallen oft erst dann auf, wenn sie bei Analysen zu Fehlern führen.

2.1.4 Verfügbare Dokumentation

Obwohl es reichlich Dokumentation zu Visual Basic 6 gibt, werden viele Aspekte, die für die Analyse von Programmen bedeutsam sind, kaum oder gar nicht erläutert. Die zur Verfügung stehenden Bücher ([Mon01, Cow00, Kof00, Mas99]) richten sich in der Regel an unerfahrene Programmierer und beschreiben die Sprache vornehmlich anhand von Beispielen und bieten keine systematische Einführung in die Konzepte der Sprache.

Eine Sprachreferenz bietet neben [Hau99] vor allem Microsofts offizielle Online-Dokumentation [Mic06b]. Leider sind auch diese Quellen unvollständig, da manche Sprachbestandteile gar nicht beschrieben werden. Andere werden dagegen mit fehlerhaften Informationen präsentiert (beispielsweise die Methoden `Play` oder `Line` für GUI-Komponenten in [Mic06b]).

Letztendlich ist allen Quellen gemein, dass sie hinsichtlich der Syntax und der Semantik unvollständig sind und die Sprache auf einem recht einsteigerfreundlichen Niveau beschreiben. Kkomplexere Hintergründe oft gar nicht erörtert werden. Sicherlich ist bei keiner Programmiersprache zu erwarten, dass die Dokumentation Fragen beantwortet, die sich vornehmlich bei der Programmanalyse stellen und weniger beim Programmieren in der Sprache selbst. Allerdings wirkt die Dokumentation wesentlich oberflächlicher als bei Sprachen wie C++ oder Java. Zusammen mit den zahlreichen Sonderfällen schafft die mangelnde Dokumentation so eine schwierige Ausgangssituation für die Analyse von Visual Basic 6 Programmen.

2.1.4.1 Grammatiken

Besonders problematisch ist das Fehlen einer vollständigen Grammatikdefinition. Zwar beinhalten die Referenzen Syntaxdefinitionen für einzelne Anweisungen, leider sind diese jedoch unvollständig, teilweise ungenau und geben keinen Aufschluss darüber, wie sich aus ihnen ein korrektes Visual Basic 6 Programm zusammensetzt. Der Grund für das Fehlen einer offiziellen Grammatik besteht darin, so teilte Microsoft auf eine direkte Nachfrage mit, dass es auch intern im Unternehmen keine einheitliche Grammatikdefinition gäbe. Unabhängig davon ob und in welcher Form Microsoft die Sprache intern spezifiziert, für die Zwecke dieser Arbeit ist keine offizielle Grammatik verfügbar. Die einzige absolute Referenz ist der Visual Basic 6 Compiler, der zwar verwendet werden kann, dessen Quellen aber natürlich unter Verschluss liegen. Es wird aber eine formale Grammatikdefinition benötigt, wenn eine Syntaxanalyse wie in [ASU86] beschrieben durchgeführt werden soll.

Zur Untersuchung von VB6-Programmen bietet sich keine Alternative zur Analyse der Quelltexte an. Zwar besteht die Möglichkeit die Programme, anstatt in nativen Code, in einen Bytecode namens *P-Code* zu übersetzen. Allerdings eignet sich dieser nicht als alleinige Untersuchungsgrundlage, da in ihm jegliche früh gebundene Bezeichner fehlen. Zudem ist die Dokumentation hier noch problematischer. Microsoft gibt in [Mic92] lediglich einige statis-

tische Daten an, denen zufolge die Ausführung auf einer Stackmachine stattfindet, die 511 unterschiedliche Opcode-Anweisungen interpretiert. Eine detailliertere Dokumentation dieses Bytecodes lässt sich jedoch nicht finden. Während undokumentierte Syntax relativ einfach zu verstehen sein sollte, sobald man Beispiele dafür gefunden hat, besteht der P-Code lediglich aus kryptischen Anweisungen, die ohne Dokumentation kaum zu verstehen sind.

In der Recherche zu dieser Arbeit konnten zwei VB6-Grammatiken gefunden werden, die von unabhängigen Autoren stammen und frei zur Verfügung stehen. Dabei handelt es sich um eine Definition für TXL[2] – ein System für Softwaretransformationen – von Jim Cordy, sowie um eine weitere im Format des Parsergenerators JavaCC[3], die von Paul Cager erstellt wurde [Cag06].

Beide Grammatiken sind zwar nicht vollständig und nicht mit der Vorgabe einer Implementierung in C++ vereinbar, da weder TXL noch JavaCC solchen Code generieren können. Allerdings versuchen sie Visual Basic 6 formal zu beschreiben und liefern wertvolles Wissen für die Erstellung einer neuen Grammatik.

Evaluierung der verfügbaren Grammatiken

Um abzuschätzen, inwieweit die beiden Grammatiken vollständig und korrekt sind, habe ich sie mit den Beispielprogrammen dieser Arbeit getestet. Dies ist ohne weiteres möglich, da beide Grammatiken in einer Form vorliegen, in der sie „out-of-the-box“ ausgeführt werden können. Dabei wird zum einen als einfaches Maß ermittelt, wieviele der Quelldateien korrekt erkannt wurden und in wievielen dagegen mindestens ein Fehler auftrat. Zum anderen werden die aufgetretenen Fehler anhand ihrer Meldungen untersucht, um festzustellen wie viele unterschiedliche Probleme aufgetreten sind. Das Ziel dieser Betrachtung besteht nicht darin, die „bessere“ Grammatik zu finden, sondern vielmehr ihre Vollständigkeit grob abzuschätzen.

Die Ergebnisse sollen Aufschluss darüber geben, ob und inwiefern eine Verfeinerung der Definitionen notwendig ist, um die in dieser Arbeit beabsichtigten Analysen durchzuführen. Die Ergebnisse dieses ersten Tests werden in Tabelle 2.1 dargestellt.[4]

Modulart (Anz.)	TXL	JavaCC (modifiziert)
Standard (386)	40 (10,36%)	28 (7,25%)
Klassen (134)	13 (9,70%)	3 (2,24%)
Formulare (347)	347 (100,00%)	7 (2,02%)
Gesamt (867)	400 (46,14%)	38 (4,38%)

Tabelle 2.1: Fehlerquote der Grammatiken über die Beispielquellen

Auffällig ist das völlige Scheitern der TXL-Variante bei Formularmodulen. Dies liegt darin begründet, dass die Besonderheiten der Formularmodul-Syntax in dieser Grammatik gar nicht berücksichtigt wurde. Offensichtlich wurde die Grammatik gar nicht zum Parsen von Formularmodulen konzipiert. Die JavaCC-Grammatik dagegen konnte mehr als 95% der Module vollständig parsen, ohne dabei einen Fehler zu erzeugen. In den übrigen Modulen traten nur acht unterschiedliche Fehler auf.

[2] TXL: `http://www.txl.ca`

[3] JavaCC: `https://javacc.dev.java.net`

[4] Die beim Test generierten Ausgaben der Parser, auf deren Grundlage diese Daten erhoben wurden, befinden sich auf der beiliegenden CD im Verzeichnis „grammartest“. Für diesen Test wurde ein trivialer Fehler in der JavaCC-Grammatik behoben, der zu einer Fehlerquote von über 70% bei den Formularmodulen führte.

Allerdings sind auch diese Zahlen mit Vorsicht zu betrachten, da in diesem Test nicht die Vollständigkeit der Grammatik selbst, sondern das Ergebnis ihrer *Implementierung*, also auch ihrer Actions, getestet wurde. Eine nähere Betrachtung der Grammatikdefinition zeigt, dass komplizierte Ausdrücke zum Teil schlicht ignoriert werden, indem der Parser in bestimmten Situationen gar nicht erst versucht, bestimmte Anweisungen, deren Syntax aus der Reihe fällt, korrekt abzuleiten. Stattdessen läuft er so lange über den Tokenstrom weiter, bis das Ende der Anweisung erreicht ist und setzt dort das Parsen fort. Dieses Vorgehen wird benutzt, um Sonderfälle, die nicht zu der ansonsten üblichen Syntax passen, nicht in der Grammatik behandeln zu müssen.

Zudem werden schon im Lexer Vereinfachungen vorgenommen, die ungünstig sind, wenn eine semantische Analyse durchgeführt werden soll. Beispielsweise kann die Deklaration `Dim i As Integer` abgekürzt werden, indem ein spezielles Kürzel zur Angabe des Typs eingesetzt wird und die Deklaration nur noch `Dim i%` lautet. Hierbei kennzeichnet das Prozentzeichen den Typ Integer und ist nicht Teil des Bezeichners. Um einerseits die Bezeichnernamen vergleichen zu können und andererseits den Typ einer Variable erkennen zu können, sollten hier also auch zwei Token erstellt werden. Dies erspart sich die JavaCC-Grammatik jedoch und muss daher auch keine Typkürzel in der Grammatik vorsehen.

Solche Verallgemeinerungen werden dort vorgenommen, da die Grammatik lediglich für Transformationen konzipiert wurde, bei denen diese Details keine Rolle spielen. Dabei ging es beispielsweise um das automatische Einfügen von Sprungmarken zur Fehlerbehandlung. Um die für den RFG benötigten Daten zu sammeln sind jedoch sehr viel genauere Analysen, die vor allem auch semantische Aspekte betrachten, notwendig. Viele der dazu notwendigen Details werden in der vorhandenen Grammatik nicht berücksichtigt. Diese Verallgemeinerungen müssen also bei der Betrachtung des obigen Testergebnisse berücksichtigt werden.

2.1.4.2 Ergebnisse

Die bedeutendste Erkenntnis aus der Suche nach vorhandenen Grammatiken und deren Untersuchung ist die Tatsache, dass keine vollständige Grammatikdefinition für Visual Basic 6 zur Verfügung steht, die eine syntaktische Analyse der Quelltexte ermöglichen, die für die Zwecke dieser Arbeit genügt. Daher ist es erforderlich eine entsprechende Grammatik selbst zu definieren. Sowohl die TXL- als auch die JavaCC-Grammatik konnten jedoch einen Großteil des Beispielcodes, auf den beide nicht zuvor angewendet wurden, korrekt erkennen. Die TXL-Variante war zwar nicht in der Lage, Formularmodule zu parsen, allerdings unterscheiden sich diese syntaktisch nur in ihren Kopfdaten wesentlich von den anderen Modularten. Es ist daher zu vermuten, dass die übrigen Bereiche der Formularmodule eine ähnliche Fehlerrate aufweisen, wie sie bei den anderen Modultypen gezählt wurde. Dort ist in etwa 10% aller Quelldateien mindestens ein Fehler aufgetreten, in allen anderen dagegen keiner. Demnach wird ein nicht unerheblicher Teil der VB6-Syntax hier erkannt. Die JavaCC-Grammatik erreicht im durchgeführten Test sogar bessere Ergebnisse.

Diese relativ oberflächliche Betrachtung erlaubt keine genauen Aussagen darüber wie nahe sich die Grammatiken tatsächlich an der Sprache befinden, allerdings ist klar, dass sie eine Sprache definieren in der ein Großteil der Beispielprogramme ebenfalls enthalten sind. Sie liefern daher wertvolles Wissen über die Syntax von Visual Basic 6, das als Ausgangspunkt für die Erstellung einer neuen Grammatik, die den Anforderungen dieser Arbeit genügt, dienen kann.

2.2 COM - Das Component Object Model

Bei der Betrachtung von Visual Basic 6 muss auch auf das ***Component Object Model***, kurz *COM*, eingegangen werden. Es ist selbst kein Bestandteil der Sprache, sondern vielmehr ein grundlegendes Konzept in der Windowsprogrammierung. Es bildet den Rahmen für die Verwendung von Bibliotheken und gemeinsam genutzten Objekten im Betriebssystem und wird auch von Visual Basic 6 intensiv genutzt. COM-Komponenten können damit angesprochen, aber auch erstellt werden. Microsoft definiert COM in [Mic06a] wie folgt:

> *„The Microsoft Component Object Model (COM) is a platform-independent, distributed, object-oriented system for creating binary software components that can interact.“*

COM ist fest mit dem Betriebssystem verbunden und verwendet die Windows-Registry, um die installierten Komponenten zu verwalten. Hierbei ist praktisch alles – sei es die Komponente selbst, ihre Schnittstellen oder sogar einzelne Methoden – durch einen eindeutigen Schlüssel gekennzeichnet. Dieser nennt sich GUID (für ***G**lobal **U**nique **Id**entifier*).

Die Verwendung von COM-Komponenten ist auf verschiedenen Wegen möglich. Für Visual Basic 6 gibt es zwei: Zum einen die direkte Ansprache eines bekannten Objekts, die Microsoft als frühes Binden bezeichnet und zum anderen den Aufruf eines Objekts unbekannten Typs, bei dem zunächst abgefragt wird, ob das Objekt die aufgerufene Methode oder das angefragte Attribut unterstützt. Im zweiten Fall findet die Bindung an die *Komponente* erst zur Laufzeit statt. Dies nennt der Hersteller spätes Binden (vgl. [Mic03, Loo01]).

2.2.1 Spätes Binden in COM

Microsofts Form der späten Bindung stellt für die RFG-Generierung in dieser Arbeit ein besonderes Problem dar. COM definiert hier eine besondere Form des Dispatchings, welches in Visual Basic 6 dazu führt, dass manche Bezeichner im Quelltext ohne Laufzeitanalyse gar nicht aufgelöst werden können. Dieses Problem werde ich im Folgenden näher erläutern.

Eine Komponente kann mehrere Klassen in sich vereinen. COM definiert ein einheitliches Prinzip, um deren Schnittstellen anzusprechen. Dazu verfügt jede Komponente über die Schnittstelle `IUnknown`. Alle Klassen müssen zudem das Interface `IDispatch` implementieren. Dabei definiert `IUnknown` Methoden, die es erlauben, die einzelnen Schnittstellen der Komponente anhand ihres Namens oder ihrer GUID abzufragen. `IDispatch` dient dazu, die einzelnen Methoden und Attribute einer bestimmten Schnittstelle anhand deren Namen oder GUIDs abzufragen und diese aufzurufen.

`IUnknown` und `IDispatch` definieren also letztendlich einheitliche Zugriffsmethoden auf die Bestandteile von Komponenten. Über sie ist es möglich, Symbole zur Laufzeit an eine beliebige Komponente zu binden, indem geprüft wird, ob diese Symbole in der Komponente vorhanden sind. Ist dies der Fall, können sie aufgerufen werden. In Visual Basic ermöglicht dies die Verwendung von Objekten, ohne dass deren Typ bekannt sein muss. Listing 2.9 zeigt eine Prozedur, der ein Objekt übergeben wird und das seine Attribute nutzt, ohne Kenntnis von seinem Typ zu haben.

```
Public Sub printObjectName (o As Object)
    Debug.Print o.Name     'kein konkreter Typ fuer o bekannt
End Sub
```

Listing 2.9: Verwendung eines Objektes ohne Kenntnis seines Typs

Was genau durch `o.Name` aufgerufen wird entscheidet sich erst zur Laufzeit. Visual Basic verwendet dabei die `IDispatch`-Schnittstelle der übergebenen Objektinstanz, um das Attribut `Name` abzufragen und anzusprechen. Dies ist das späte Binden im Sinne des Component Object Models. Hier herrscht Polymorphie nicht nur, wie sonst in vielen Sprachen üblich, innerhalb einer Vererbungshierarchie, sondern über alle möglichen COM-Objekte. Der Visual Basic Compiler kann in diesem Fall keine Bindung vornehmen, da schlicht gar kein Typ bekannt ist. Sollte in obigem Beispiel ein Objekt übergeben werden, das über gar kein `Name`-Attribut verfügt, so führt dies zu einem Laufzeitfehler.

Dieses Verhalten ist nicht nur auf den speziellen Datentypen `Object` beschränkt. Es kann auch mit dem Typen `Variant`, der zusätzlich auch primitive Typen repräsentieren kann, oder sogar konkreteren Typen, wie beispielsweise `Form` oder `Control`, angewendet werden. Die spezifischen Attribute und Methoden eines Formulars können beispielsweise auch dann angesprochen werden, wenn das Formular in einer Variable vom generelleren Typ `Form` referenziert wird – auch hier findet das spät bindende COM-Dispatching statt. Also sogar innerhalb eines einzelnen VB6-Programms.

Durch statische Analysen allein lässt sich wegen dieser Form des späten Bindens nicht immer eindeutig feststellen von welchem Typ ein Bezeichner ist. Das hat zur Folge, dass es auch nicht in jedem Fall möglich ist, alle für den RFG benötigten Informationen statisch zu ermitteln. Zwar muss jedes Objekt am Beginn seines Lebenszyklus als Instanz einer konkreten Klasse erstellt werden, jedoch muss dies nicht notwendigerweise auch im verfügbaren Quelltext passieren. COM-Komponenten können selbst Attribute oder Methoden enthalten, die generische Typen, also beispielsweise `Objects`, zurückgeben. In diesem Fall ist die Information, von welchem Typ ein Objekt tatsächlich ist, schlichtweg nicht vorhanden.

Auch die Instanziierung von Objekten kann in Visual Basic über die `IUnknown`- und `IDispatch`-Schnittstellen vorgenommen werden. Dies ist sogar notwendig, wenn die COM-Komponente keine *Typelib* – eine Auflistung aller Schnittstellen in binärem Format – bereitstellt. Listing 2.10 zeigt eine solche Initialisierung. Hierbei dient Visual Basics `createObject`-Methode als Wrapper um den komplexen Aufruf über die `IUnknown`-Schnittstelle der Komponente namens `Komponente`.

```
Dim o As Object
Set o = createObject("Komponente.Klasse")
```

Listing 2.10: Instanziierung einer Bibliotheksklasse ohne Typelib

Somit gibt es also Fälle, in denen es Hinweise auf den Typ der spät gebundenen Klassen gibt, aber auch andere Situationen, in denen diese Information vollkommen fehlt. Es gibt unterschiedliche Methoden, um das Ziel eines Zeigers und spät gebundene Aufrufe statisch zu analysieren. Robert Wilson befasst sich in [Wil99] unter anderem mit dem Problem der Zeigeranalyse im Allgemeinen. Er nennt unterschiedliche Verfahren, die dazu dienen, Funktionszeiger oder virtuelle Methodenaufrufe zu analysieren, um zu ermitteln, worauf diese zeigen können. Allerdings lassen sich diese nur sehr bedingt zur Betrachtung des späten Bindens im Component Object Model heranziehen.

Um derartige Analysen durchführen zu können, ist es erforderlich, die möglichen Ziele der Zeiger zu kennen. Beispielsweise indem die möglichen Werte, die ein Funktionszeiger annehmen kann, oder die Spezialisierungen einer virtuellen Methode innerhalb einer Vererbungshierarchie gesammelt werden. Die Menge der möglichen Ziele lässt sich anhand von Analysen einengen, indem einzelne Ziele aufgrund unterschiedlicher Kriterien ausgeschlossen werden.

Das Ermitteln aller möglichen Ziele ist in Visual Basic allerdings unter Umständen gar nicht möglich, da der Typ von verwendeten Objekten gänzlich unbekannt sein kann. Zudem sind

die Schnittstellen von Komponenten ohne Typelib unbekannt. Daher fehlen unabdingbare Ausgangs- und Vergleichsdaten für Zeigeranalysen, so dass sie nicht angewendet werden können.

Eine Instanziierung wie in Listing 2.10 gibt aber Aufschluss über den Typ der erstellten Instanz. Eine Kontrollfluss-sensitive Analyse könnte mit dieser Information nachvollziehen, welcher Typ sich tatsächlich hinter einer `Object`-Variable verbirgt. Allerdings ist auch eine solche Betrachtung wenig vielversprechend. Visual Basic kennt nämlich keine Cast-Operation für nicht-primitive Typen und erlaubt lediglich den Typ einer Objektreferenz mittels Aliasing – sprich der Zuweisung der Objektreferenz zu einer Variable eines allgemeineren Typs – zu generalisieren. Sobald ein Zeiger zum Typ `Object` verallgemeinert wurde, kann er nicht wieder konkretisiert werden. Funktionen, die nie einem Zeiger zugewiesen werden, kommen beispielsweise nicht als Kandidat für das Ziel eines Funktionszeigers in Frage.

Sobald `Object`-Referenzen an Funktionen übergeben werden, ist damit zu rechnen, dass die Menge an möglichen Typen der `Object`-Parameter steigt. Sie wären nicht als `Object` definiert worden, würden nicht unterschiedliche Typen erwartet. Ab dieser Stelle kämen im konkreten Fall also sehr wahrscheinlich mehrere Typen in Frage, wobei im Folgenden keine Konkretisierungen des Zeigertyps mehr vorgenommen werden. Es ist zwar möglich, dass eine Zeigeranalyse unter diesen Bedingungen erfolgreich verläuft, allerdings steht das zu erwartende Ergebnis in keinem Verhältnis zum Aufwand, den die Konzeption und Implementierung eines solchen Verfahrens erfordern würde.

Es gibt jedoch auch sehr einfache Verfahren, die im Allgemeinen zwar nur geringen Erfolg versprechen, bei Visual Basic wegen einiger häufig auftretenden Programmierpraxis dennoch interessant erscheinen. Oft ist zu sehen, dass eine COM-Komponente über die `createObject`-Funktion als `Object`-Referenz initialisiert, dann aber nur zum Aufruf von Bibliotheksfunktionen verwendet und wieder verworfen wird. In solchen Fällen, in denen eine Referenz nur einmalig gesetzt wird, steht der Typ des Objektes durch den Parameter der `createObject` fest.

Dann sind zumindest die Namen von Komponente und Typ klar. Allerdings nutzen diese Informationen nur dann etwas, wenn die Schnittstelle der Komponente auch bekannt ist, sprich wenn eine Typelib vorhanden ist. Objekte, die über `createObject` erzeugt werden, dürften in der Regel aber keine Typelib haben, sonst hätte ja die performantere frühe Bindung verwendet werden können. Viel gewonnen wäre in diesem Fall nicht, da die wenig strikte Syntax in den seltensten Fällen Aufschluss darüber gibt, ob sich hinter einem Bezeichner eine Prozedur, Property oder Variable verbirgt. Klar wäre nur, dass ein Symbol irgendeiner Art mit einem bestimmten Namen, das zu einer nicht weiter bekannten Komponente und Klasse gehört, in irgendeiner Weise referenziert wird.

Beim späten Binden durch COM-Dispatching existieren also Methoden, die dabei helfen können, in gewissen Fällen den tatsächlichen Typ von Objekten und die Identität von aufgerufenen Methoden und verwendeten Attributen zu erkennen. Der Aufwand, der betrieben werden müsste, um eine nennenswerte Verbesserung der Analyseergebnisse zu erreichen, ist wie gesagt aber unangemessen hoch. Zudem ginge der Einsatz komplexer Methoden über den Rahmen dieser Diplomarbeit, die zunächst einmal Visual Basic syntaktisch analysieren können muss, hinaus. Daher werde ich in dieser Arbeit keine weitere Analyse dieser COM-Besonderheit durchführen.

Die eben behandelten Probleme bestehen nicht bei jeder Verwendung von COM-Komponenten. Wie die Existenz einer späten Bindung nahelegt, existiert auch ein frühes Binden. Dieses kann stattfinden, wenn die Komponente über eine Typelib verfügt. Üblicherweise verfügen COM-Komponenten über so genannte *duale Schnittstellen*, die sowohl frühes, als auch spätes Binden

ermöglichen. Letzteres wird benötigt, um Klassen überhaupt mit den `Object`-Datentypen verwenden zu können. Alle mit Visual Basic 6 erstellten Komponenten verfügen von Haus aus über eine solche duale Schnittstelle.

Da das frühe Binden wesentlich schneller erfolgt, kann davon ausgegangen werden, dass es in der Regel dem COM-Dispatching vorgezogen wird. Wie ich im Verlauf der Arbeit noch erläutern werde, stellt das späte Binden die Ausnahme dar und beeinträchtigt die Analyseergebnisse daher nur in begrenztem Maße.

2.2.2 Typbibliotheken

Die Schnittstellen von COM-Komponenten müssen für die semantische Analyse von VB6-Quelltexten aus zwei Gründen bekannt sein. Zum einen wird eine Namensauflösung betrieben, in der unter anderem qualifizierte Bezeichner ihren Symbolen zugeordnet werden. Da die Elemente eines qualifizierten Bezeichners nur dann korrekt aufgelöst werden können, wenn der Typ des vorhergehenden Elements bekannt ist, muss dieses Wissen über die von den Bibliotheken exportierten Symbole bekannt sein. Der andere Grund ergibt sich aus der Möglichkeit, lokale Variablen in Visual Basic 6 implizit durch deren Verwendung zu deklarieren. Zugleich können Bibliotheken aber auch Symbole global im Programm verfügbar machen. Daher lassen sich implizit deklarierte Variablen nur dann von global verfügbaren Bibliothekssymbolen unterscheiden, wenn die Symbole der Bibliotheken auch bekannt sind. Wenn vorhanden, müssen die Typelibs der COM-Komponenten daher ausgelesen werden. (In Bezug auf implizit deklarierte Variablen stellen Komponenten ohne Typelibs kein Problem dar, da diese keine Symbole global verfügbar machen können.)

Da die Komponenten oft nur anhand ihrer GUID, die nur über die Windows-Registry einer Komponente zugeordnet werden kann, referenziert werden und Werkzeuge zum Auslesen von Typelibs nur auf Windows-Systemen lauffähig sind, muss diese Untersuchung unter Windows erfolgen. Die Entwicklung der Analysewerkzeuge findet aber unter Linux statt (siehe Abschnitt 1.2). Daher muss eine Brücke zwischen dem binären Format unter Windows und den linuxbasierten Analysewerkzeugen geschlagen werden. Die dazu erarbeitete Lösung beschreibe ich in Abschnitt 4.2.

2.3 Metamodelle zur Programmrepräsentation

Softwaresysteme lassen sich auf verschiedene Arten repräsentieren. Die ureigene Form ist der Programmquelltext, der die konkrete Realisierung darstellt. Allerdings ist er oft kein angemessenes Mittel, um Softwaresysteme zu betrachten. Er ist zu konkret und oft auch zu komplex, um einen Überblick zu bieten. Ebenso ist es möglich, dass er in der Planungphase eines Softwaresystems noch gar nicht existiert. In solchen Fällen bietet es sich an, eine Abstraktion vom tatsächlichen Gegenstand oder von einem Problem in Form eines Modells zu schaffen und stattdessen dieses zu betrachten und zu entwickeln. Aspekte des Softwaresystems, die für eine bestimmte Betrachtung nicht relevant sind, lassen sich so ausblenden, vereinfachen oder verallgemeinern. Ebenso können Modelle auch dazu dienen, eine Basis zu schaffen, um unterschiedliche Dinge miteinander zu vergleichen und auf einheitliche Art und Weise handzuhaben.

Ein Metamodell stellt ein Schema dar, nach dem Modelle für konkrete Phänomene der realen Welt auf einheitliche Weise repräsentiert werden können. In diesem Abschnitt stelle ich einige Metamodelle vor, die im Reengineering verwendet werden, um Programme auf einer Abstraktionsstufe darzustellen, die im Wesentlichen globale Programmbestandteile wie Variablen, Funktionen, Klassen und dergleichen sowie deren Beziehungen beinhaltet. Zum einen handelt es sich dabei um den *Resource Flow Graph* aus dem Bauhaus-Projekt, der in dieser Arbeit für Visual Basic 6 Programme erstellt werden soll. Daher ist seine Beschreibung der Schwerpunkt. Zusätzlich werde ich mit dem *Famix-Model* und dem *Dagstuhl Middle Metamodel* zwei ähnliche Metamodelle kurz vorstellen und vergleichen.

2.3.1 Der Resource Flow Graph

Der Resource Flow Graph (RFG) (siehe [RVP06, EKP+99]) dient im Bauhaus-Projekt zur Darstellung und Untersuchung von Softwaresystemen auf architektonischer Ebene. Er kann mit dem Programm *Gravis* visualisiert und manipuliert werden. Im Gegensatz zur wesentlich komplexeren und quelltextnahen Intermediate Language (IML) ermöglicht er es dem Betrachter, einen Überblick der architektonisch relevanten Bestandteile und ihrer Zusammenhänge zu gewinnen.

Ein RFG ist ein gerichteter Graph, dessen Knoten die globalen Bestandteile eines Softwaresystems darstellen. Hierfür existieren abstrakte Konzepte, die sich dazu eignen, Programme unabhängig von ihrer Implementierungssprache darzustellen – beispielsweise Routinen, Typen, Klassen oder Module. Die Beziehungen zwischen diesen Elementen werden durch die Kanten des Graphen dargestellt. Sie repräsentieren die globalen Zusammenhänge wie Aufrufe oder die Typzugehörigkeit eines Elements. Knoten und Kanten sind dabei typisiert und lassen sich in zwei Gruppen einteilen, die folgenden Zwecken dienen:

1. Abbildung von Elementen und Beziehungen, die aus dem Quelltext hervorgehen. Aus den Quelltexten abgeleitete Basis-RFGs bestehen ausschließlich aus diesen Konzepten.

2. Konzepte, die zur Darstellung übergeordneter, architektonischer Zusammenhänge dienen. Diese werden dem Basis-RFG durch Analysen oder durch manuelle Bearbeitung des Anwenders hinzugefügt.

Die unterschiedlichen Knoten- und Kantentypen des Metamodells sind in Hierarchien organisiert, die unter anderem auch die obige Einteilung abbilden. Abbildung 2.1 zeigt einen Teil des RFG-Modells, nämlich den, der für die Modellierung von Programmen der Sprache C zur

Verwendung kommt, in einer UML-Notation. Hierbei repräsentieren die Klassen-Entitäten die Knotentypen und deren Vererbungshierarchie, während die Assoziationen die Kantentypen des RFG repräsentieren. Im Diagramm verbinden sie die Knotentypen, deren Instanzen sie in konkreten RFGs miteinander verknüpfen können. Dabei repräsentieren die Pfeilspitzen die Richtung dieser Beziehung. Im Gegensatz zu den Knotentypen kann ein Kantentyp mehrmals in der Darstellung enthalten sein, um unterschiedliche Verbindungsmöglichkeiten darzustellen. Daher sei angemerkt, dass es sich bei namensgleichen Kanten stets um den gleichen Kantentyp handelt. Ebenso muss darauf hingewiesen werden, dass einzelne Assoziationen, die im Diagramm mit mehreren Namen versehen sind, als zusammenfassende Darstellung für alle einzelnen Kanten zu verstehen sind. Das Schema abstrahiert zudem von den generelleren Typen des RFG und stellt gezielt nur jene dar, die zur Abbildung der Sprache tatsächlich zur Verwendung kommen. Einzige Ausnahme sind Generalisierungen, die die Diagrammdarstellung vereinfachen, wie hier im Falle des `Object` Knotentyps. Das Diagramm stellt also im Wesentlichen eine sprachrelevante Teilsicht auf das gesamte RFG-Modell dar, die im übrigen auch auf die Darstellung obligatorischer Attribute wie den Namen einer Entität oder ihrer Position im Quelltext verzichtet. Das Diagramm ist hier in der Originalform abgebildet, in der es in der Bauhaus-Dokumentation enthalten ist. In der dortigen Darstellung wurde auf die Angabe von Multiplizitäten verzichtet.

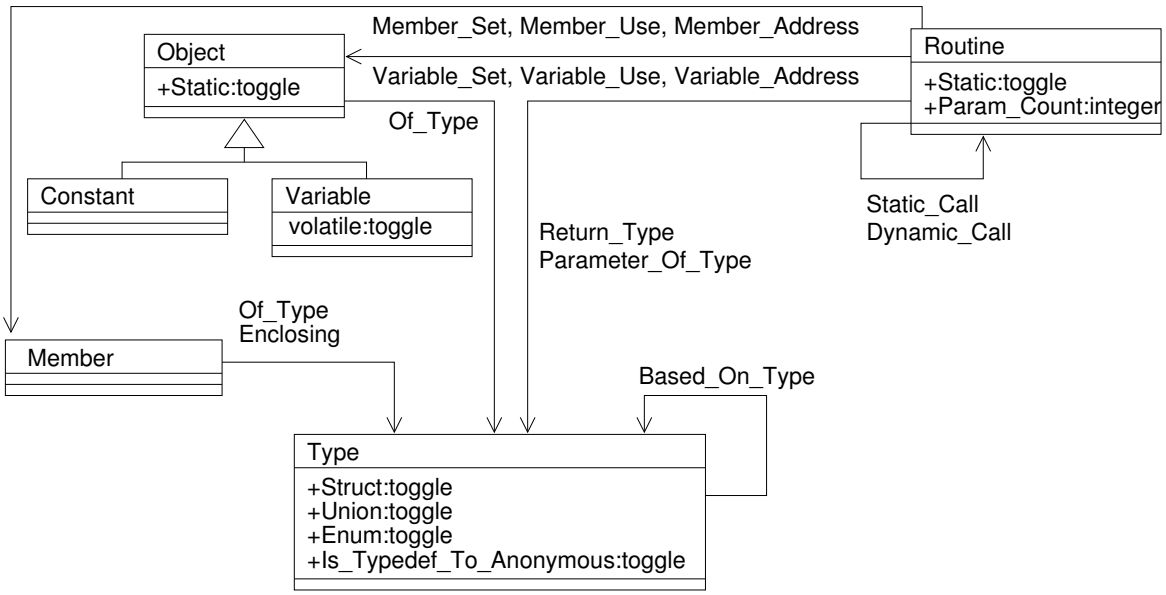

Abbildung 2.1: Das RFG-Schema für C

Der Knotentyp `Type`, `Routine`, die von `Object` abgeleiteten `Constant` und `Variable` sowie ihre Beziehungen in Form von Kanten modellieren die globalen Elemente der C Programme. Vererbungshierarchien sind in der Implementierung des RFG direkt in Form von Ada-Klassen umgesetzt.

Das Schema verdeutlicht, dass die Repräsentation auf einer konzeptionellen Ebene stattfindet, die von Details des Quelltextes abstrahiert. Dies zeigt sich insbesondere darin, dass die C-Sprachelemente `Struct`, `Union` und `Enum` als `Type` abgebildet werden. Die Differenzierung des tatsächlich zugrundeliegenden Sprachelements wird lediglich durch boolesche Attribute repräsentiert, jedoch nicht durch die Struktur aus Knoten und Kanten im RFG. Die Attribute werden zudem verwendet, um zusätzliche Informationen an den RFG zu annotieren, die durchaus rein informativen Zwecken dienen können. Das `static`-Attribut des `Object` Knotentyps, dass das `static`-Schlüsselwort aus C modelliert, zeigt zudem, dass die Beschränkung auf

globale Elemente nicht bedeutet, dass nur solche abgebildet werden, die auch wirklich global sichtbar sind. Vielmehr beinhaltet der RFG die Programmelemente, die auf der Ebene globaler Definitionen deklariert wurden, ungeachtet ihrer Sichtbarkeit. Dies bedeutet beispielsweise, dass auch private Variablen eines Programms Teil des RFG sind, solange sie auch als global sichtbare Elemente auftreten könnten. Rein strukturelle Elemente wie Module oder Packages sind nicht im Schema abgebildet.

Während die Knoten und ihre Attribute Informationen darstellen, die aus den Deklarationen des Programms hervorgehen, stellen die Kanten eines RFG semantische Informationen dar. Sie geben insbesondere Aufschluss darüber, welche Routinen auf welche anderen Programmelemente zugreifen und wie das geschieht - beispielsweise lesend oder schreibend auf einer globalen Variable. Aufrufe von Routinen können statisch oder dynamisch erfolgen. Hier wird zwischen im Quelltext sichtbaren Aufrufen und solchen über Funktionszeiger unterschieden. Diese Kanten repräsentieren Informationen, die aus einer Programmebene stammen, die der RFG selbst gar nicht abbildet, nämlich dem lokalen Bereich von Routinen. Somit bildet der RFG zwar nur global relevante Elemente und Beziehungen ab, deren Existenz kann allerdings durchaus von Programmteilen ausgehen, die selbst gar nicht im Detail abgebildet sind.

Die Bedeutung der weiteren Kantentypen geht aus ihrer Bezeichnung hervor. Es sei lediglich angemerkt, dass die `Enclosing`-Kante, die die Zugehörigkeit zu einem Typen modelliert, nicht als Verb, sondern als Adjektiv zu lesen ist. Member werden also von Typen umgeben, nicht umgekehrt.

Ein wesentliches Merkmal des RFG ist seine Graphstruktur, die die Grundlage für viele Analysen darstellt. Viele semantische Aspekte eines Programms lassen sich durch das durchlaufen bestimmter Teilgraphen untersuchen. Beispielsweise bilden die `Call`-Kanten zusammen mit den Knoten, die sie verbinden, den Aufrufgraphen des Programms. Um diese unterschiedlichen Aspekte sichtbar zu machen und Analysen auf bestimmte Teilgraphen beschränken zu können, ist es möglich, Teilsichten des Graphen zu definieren und diese in Gravis darzustellen. Das aus den Quelltexten extrahierte RFG-Modell wird dabei in der so genannten *Base-View* dargestellt. Andere typische Anwendungen für das Sichtkonzept sind:

- **Environment-View:** Enthält alle Knoten, die nicht im analysierten Quelltext definiert wurden, sprich aus der *Umgebung* des Programms stammen.

- **Module-View:** Ordnet die Programmbestandteile nach ihrer physikalischen Zusammengehörigkeit in Module – beispielsweise gemäß ihrer Aufteilung in Quelltextdateien – ein.

- **Call-View:** Der Aufrufgraph des Programms.

Darüber hinaus lassen sich eigene Sichten nach beliebigen Kriterien erstellen. Viele Analysen liefern zudem neue Sichten auf den Graphen als Ergebnis.

Zusammenfassend ist ein RFG ein Modell für die globalen Elemente von Quelltexten, das sich durch Elemente einer höheren Abstraktionsstufe, die die Architektur wiederspiegeln, anreichern lässt. Im folgenden Kapitel werde ich ein Schema entwickeln, mit dem sich Visual Basic 6 Programme auf das RFG-Metamodell abbilden lassen. Zunächst werde ich aber einen kurzen Ausblick auf andere Metamodelle, die im Reengineering verwendet werden, geben.

2.3.2 Das FAMIX-Modell

An der Universität Bern beschäftigte sich das FAMOOS-Projekt[5] von 1996 bis 1999 mit Reengineering-Methoden für objektorientierte Softwaresysteme. In seinem Rahmen wurde das *FAMOOS Information Exchange Model*, kurz *FAMIX*, entwickelt, das Programmquelltexte auf einer ähnlichen Abstraktionsstufe wie der RFG modelliert. Beschrieben wird es unter anderem in [Tic01]. Die Intention ist es, im gesamten Reengineering-Prozess und allen daran beteiligten Werkzeugen ein einheitliches Modell zu verwenden, das den Anforderungen aller dabei anfallenden Aufgaben gerecht wird. Daher ist die Repräsentationsform des Metamodells so gewählt, dass der Austausch der abgebildeten Informationen durch Standardformate vereinfacht wird.

Auch das FAMIX-Modell besteht aus einer Hierarchie von Konzepten und stellt Entitäten sowie deren Beziehungen dar. Die dabei dargestellten Konzepte überdecken sich zu einem großen Teil mit denen vom RFG. Allerdings modelliert FAMIX zum Teil auf einer Ebene, die dem Quelltext näher ist. Während des RFG beispielsweise nur die reine Tatsache darstellt, dass eine Funktion eine andere aufruft, beinhaltet ein FAMIX-Modell sehr konkrete Informationen über jeden einzelnen Aufruf und dessen Argumente. Ebenso stellt es auch lokale Variablen dar. Dieses geringere Abstraktionsniveau dürfte sich dadurch erklären, dass das vorgesehene Einsatzspektrum für dieses Metamodell größer ist. Während der RFG vornehmlich mit der Intention geschaffen wurde, das Programmverstehen zu unterstützen, soll FAMIX ein einheitliches Datenformat für gesamten Reengineering-Prozess bieten und muss daher beispielsweise auch detaillierte Informationen für Refractorings des Quelltextes beinhalten. Zugleich enthält seine Dokumentation keine Hinweise auf abstrakte Konzepte, die über die Darstellung von Quelltextelementen hinausgehen.

Anders als der RFG bildet es keinen Graphen, der es erlaubt, alle Beziehungen auf einheitliche Weise über die Kanten nachzuvollziehen. Zwar werden die Informationen, die in den Basis-RFGs enthalten sind, überwiegend auch in entsprechenden FAMIX-Modellen repräsentiert, jedoch in unterschiedlicher Form. So gibt es beispielsweise kein eigenes Konzept für Typen. Stattessen wird die Typzugehörigkeit eines Elements als textuelles Attribut dargestellt. Auf der anderen Seite werden Aufrufe und Variablenverwendungen durch Konzepte modelliert, die einer Kante im Graph sehr ähnlich sind.

FAMIX wurde konzipiert, um den Prozess von einem Programmquelltext hin zu einem geänderten Quelltext zu begleiten und bleibt daher konzeptionell näher an diesem. Der RFG entsteht dagegen aus Quelltexten und verfolgt primär das Ziel, von deren Detail zu abstrahieren und globale Betrachtungen zu ermöglichen. Er ist also auf Prozesse ausgerichtet, in denen der Abstraktionsgrad vom eigentlichen Programm erhöht wird.

Dabei ist allerdings anzumerken, dass FAMIX Möglichkeiten zur Erweiterung seines Modells bietet und sich daher auch für Betrachtungen auf einem höheren Abstraktionsniveau eignen kann. Lediglich das Grundmodell bleibt fest vorgegeben und bildet eine Basis, um Informationen über ein Softwaresystem auch dann zwischen verschiedenen Werkzeugen auszutauschen, wenn diese unterschiedliche Erweiterungen des Modells vornehmen.

2.3.3 Das Dagstuhl Middle Metamodel

In [LTP04] wird das *Dagstuhl Middle Metamodel* (DMM) beschrieben. Es ist aus dem Bestreben erwachsen, ein allgemein akzeptiertes Metamodell zu entwickeln, das einen Austausch von Modellen zwischen möglichst vielen Anwendungen im Reengineering ermöglicht. Entstanden ist es in einer internationalen Zusammenarbeit, an der auch die Initiatoren der beiden zuvor

[5] der Name steht für *Framework-based Approach for Mastering Object-Oriented Software Evolution*

beschriebenen Metamodelle mitgewirkt haben.

Im wesentlichen folgt es der Intention von FAMIX ein Austauschformat bereitzustellen, dass einen gemeinsamen Nenner von Konzepten definiert und darüber hinaus erlaubt, individuelle Erweiterungen von beliebigem Abstraktionsgrad hinzuzufügen. Das Streben nach einem gemeinsamen Format macht sich im DMM bemerkbar, indem die abstrakteren Architekturkonzepte des RFG ebenso fehlen wie die quelltextnäheren Modellierungen des FAMIX-Modells, das ich zuvor beschrieben habe. Die wesentliche Vorlage für die Beziehungen, die das Modell darstellt bietet die Unified Modeling Language (UML) (vgl. [RJB99]), deren Assoziationsklassen übernommen wurden.

Ein wesentlicher Unterschied zu den bisherigen Modellen besteht darin, dass das DMM neben dem eigentlichen Modell der Softwaresystems zusätzlich über eine (optionale) Schicht verfügt, die es ermöglicht auch die Struktur der Quelltexte zu repräsentieren und die Konzepte des Modells in diese einzuordnen. Hierdurch soll es ermöglicht werden, das Programm-Modell näher an die eigentliche Quelltext-Repräsentation des Programms anzubinden. Zudem soll es dadurch insbesondere Reverse- und Reengineeringtools ermöglichen, die Zuordnung von Modellkonzepten zu Quelltexten zu erleichtern.

2.4 Software-Metriken

Die Aufgabe dieser Arbeit schließt neben der Definition und Generierung des RFG für Visual Basic 6 auch das Erheben von Metriken über die untersuchten Programme und deren Annotierung an den RFG ein. Daher soll der Begriff der Software-Metrik, sowie die zur Aufgabenstellung gehörenden Metriken an dieser Stelle kurz vorgestellt werden. Die Erhebung der Metriken werde ich in Kapitel 4 beschreiben.

Ian Summerville definiert den Begriff der Software-Metrik in [Som04, S. 655] folgendermaßen:

> *„A software metric is any type of measurement which relates to a software system, process or related documentation.“*

Als die drei grundlegenden Ziele dieser Messungen geben Fenton und Pfleeger in [FP96, S. 11ff] das Verstehen von Entwicklungs- und Wartungsprozessen, die Kontrolle von Projekten und die Verbesserung von Prozessen und Produkten an. Diese sollen erreicht werden indem der gegenwärtige Zustand der Software festgestellt und die Einhaltung vorgegebener Kriterien überwacht werden. Hierzu lassen sich viele Aspekte eines Softwaresystems und seines Umfeldes betrachten, die dabei helfen Aufschluss über Kosten, Aufwand, Produktivität, Zuverlässigkeit oder die Performanz zu erlangen. Der Vergleich mit Erfahrungswerten aus anderen Projekten kann zudem dazu verwendet werden, Voraussagen über die weitere Entwicklung eines Softwareprojektes anzustellen.

Die Messungen teilt [FP96] in die drei Klassen *prozess-*, *produkt-* und *ressourcenbezogener* Metriken ein. Alle drei werden nochmals in die Untersuchung interner und externer Attribute unterteilt, wobei sich nur die internen Produktattribute ausschließlich mit den Programmen selbst als einzigen Gegenstand befassen, ohne äußere Einwirkungen einzubeziehen. Da in dieser Arbeit lediglich Programmquelltexte untersucht werden, sind hier nur solche Metriken von Interesse. Diese unterteilen sich nochmals in zwei Kategorien:

- **Größenmetriken**, die Aspekte wie die Länge, Funktionalität oder Komplexität betrachten.
- **Strukturmetriken**, die beispielsweise den Kontrollfluss, Datenstrukturen oder objektorientierte Aspekte analysieren.

In der Aufgabenstellung dieser Arbeit sind drei Metriken explizit vorgegeben. Dies sind die Größenmetriken *Lines of Code* und *Halstead* sowie die Strukturmetrik der *zyklomatischen Komplexität*. Alle drei werden im Folgenden vorgestellt.

2.4.1 Lines of Code

Eine der einfachsten Methoden die physikalische Länge eines Programms zu messen ist, die Anzahl der Zeilen aus denen der Quelltext besteht zu zählen. Problematisch hierbei ist jedoch, dass nicht alle Zeilen als gleichwertig betrachtet werden können. Programme bestehen typischer Weise nicht nur aus Zeilen einzelner Anweisungen, sondern zugleich auch aus Leer- und Kommentarzeilen, die keinen Einfluss auf das übersetzte Programm haben. Auch Deklarationen oder spezielle Sprachfeatures wie das Einbinden externer Dateien belegen in der Regel Zeilen, ohne jedoch ausführbarer Teil der implementierten Algorithmen zu sein. Zugleich kann eine Zeile nicht nur eine, sondern gleich mehrere Anweisungen enthalten. [FP96] nennt zudem eine Vielzahl weiterer Kriterien wie die Art auf die der Quelltext erstellt wurde

(handgeschrieben oder generiert) oder die Tatsache, ob der Code neu oder wiederverwendet ist.

Demnach hängt es von der Betrachtungsweise ab, welche Zeilen gezählt und welche ignoriert werden. Beispielsweise sind Leerzeilen zur Abschätzung des Programmieraufwands praktisch unerheblich, während das Verfassen von Kommentaren durchaus einen gewissen Aufwand bedeutet. Bei der Betrachtung von *Lines of Code* oder kurz *LOC* muss demnach genau festgelegt werden, aus welchen Zeilen sich diese Zahl genau zusammensetzt, denn unterschiedliche Messverfahren machen die Ergebnisse schlecht vergleichbar.

In [FP96, S. 246ff] werden folgende grundlegende Definitionen vorgeschlagen: Unter *LOC* werden alle Programmzeilen verstanden, die nicht leer sind. Die *Non-commented Lines of Code (NCLOC)* bzw. *Effective Lines of Code (ELOC)* messen dagegen alle Zeilen, die weder leer noch reine Kommentare sind, während die Anzahl der reinen Kommentarzeilen als *CLOC* vorgeschlagen wird. Demnach ergibt sich LOC aus der Summe von ELOC und CLOC. Zur Berechnung der Kommentardichte in einem Programm kann der Quotient:

$$\frac{CLOC}{LOC}$$

dienen.

Im Bauhaus-Projekt werden dagegen leicht andere Kriterien zur Messung herangezogen, die aus [Axi06] hervorgehen. Diese Arbeit wird sich zur Einhaltung der Konsistenz der Darstellungen unterschiedlicher Sprachen, an diese Messverfahren halten. Sie definieren die folgenden Maße für Routinen:

- **LOC**: Totale Anzahl aller Zeilen ungeachtet ihres Inhalts.
- **Empty**: Anzahl der Zeilen, die leer sind oder nur Leerzeichen enthalten.
- **Code**: Anzahl der Zeilen, die mindestens eine Anweisung der Sprache enthalten.
- **Comment**: Anzahl der Zeilen, die auch Kommentare enthalten.
- **Only_Comment**: Anzahl der Zeilen, die ausschließlich Kommentare enthalten.

Zudem kann jeweils die Auftretungshäufigkeit von Zeilen mit einer der Zeichenketten „FIXME“, „HACK“ und „TODO“ gezählt werden. Die Aufgabenstellung dieser Arbeit sieht die Erhebung der in obiger Auflistung aufgeführten Maße vor.

Generell kann Lines of Code lediglich Aussagen über die physikalische Länge eines Programmquelltextes liefern, nicht aber über die Komplexität des durch die gezählten Zeilen repräsentierten Programms. Die Metrik lässt sich jedoch sehr unkompliziert erheben und kann zur einfachen Betrachtung der Größe und dessen zeitlicher Entwicklung in Projekten, sowie als Hinweis auf die Produktivität, beispielsweise im Sinne der von einem Programmierer über einen bestimmten Zeitraum erstellten Zeilen, angewendet werden. [FP96]

2.4.2 Halstead

Maurice Halstead veröffentlichte bereits 1976 eine Metrik, die sowohl die Größe als auch Aspekte der Komplexität unter Verwendung von interdisziplinären Methoden darstellen sollte. Hierbei wird insbesondere versucht Regeln und Gesetzmäßigkeiten aus der anderen Wissenschaften, insbesondere aus dem Bereich der Psychologie, auf Software zu übertragen. Ein

Programm P wird hierzu zunächst in einzelne Token zerlegt, die jeweils als Operator oder Operand klassifiziert werden. Als Klassifizierung bietet es sich an, alle Bezeichner und konstanten Werte als Operanden, alles weitere als Operatoren zu verstehen.

Über diese Token werden dann die folgenden grundlegenden Metriken erhoben:

$$\mu_1 = \text{Anzahl unterschiedlicher Operatoren}$$
$$\mu_2 = \text{Anzahl unterschiedlicher Operanden}$$

$$N_1 = \text{Gesamtsumme verwendeter Operatoren}$$
$$N_2 = \text{Gesamtsumme verwendeter Operanden}$$

Aus diesen Daten lassen sich das *Vokabular* μ und die *Länge* N von P errechnen:

$$\mu = \mu_1 + \mu_2$$
$$N = N_1 + N2$$

Diese Werte dienen dann der Berechnung des *Volumens* V von P, dass sich wie folgt definiert:

$$V = N \times \log_2 \mu$$

Dieser Wert bezieht sich laut [FP96, S. 250f] im ursprünglichen Sinne auf die Anzahl der geistigen Unterscheidungen, die notwendig sind, um eine Programm der Länge N zu schreiben. Weiterhin lassen sich der so genannte *Program Level* L und der für P benötigte *Programmieraufwand* E abschätzen. Um die Tatsache zu unterstreichen, dass diese Werte als Schätzungen zu verstehen sind, werden sie im folgenden mit dem Zusatz *est* versehen.

$$L_{est} = \frac{2}{\mu_1} \times \frac{\mu_2}{N_2}$$
$$E_{est} = \frac{V}{L_{est}}$$

Darüber hinaus soll sich der Zeitaufwand T zur Programmierung von P aufgrund psychologischer Beobachtungen durch

$$T_{est} = E_{est}/18 Sekunden$$

berechnen lassen.

Den psychologischen Hintergründen dieser Berechnungen wird laut [FP96, S. 252] heute keine Bedeutung mehr zugemessen. Dennoch liefern die Länge N und das Volumen V sinnvolle Messgrößen, wobei das Volumen als ein Wert für den Speicherbedarf einer binären Kodierung des Programms verwendet wird. Somit sind diese Metriken in der Praxis als Größenmetriken dienlich, die jedoch keine Aussagen über die Komplexität des Programmablaufs erlauben. In dieser Arbeit ist daher lediglich die Berechnung des Volumens Teil der Aufgabe. [FP96]

2.4.3 Zyklomatische Komplexität

Thomas McCabe schlug 1976 vor, die Komplexität eines Programms anhand seines Kontrollflussgraphen zu ermitteln. Dazu wird die *zyklomatische Zahl* v eines Kontrollflussgraphen F wie folgt ermittelt (siehe [WM96]):

$$v(F) = e - n + 2$$

wobei e die Anzahl aller Kanten und n die Anzahl aller Knoten in F beschreibt. Diese zyklomatische Zahl repräsentiert die linear unabhängigen Pfade im Graphen. Die gleiche Berechnung lässt sich folgendermaßen auch ohne einen Graphen anhand des Quelltextes ausführen:

$$v(F) = 1 + d$$

Hierbei ist d die Anzahl der Binärverzweigungen im Graphen, bzw. die Anzahl der Entscheidungspunkte im Programmquelltext. Daher kann d - beginnend bei Null - an jeder Binärverzweigung um den Wert 1 erhöht werden. Bewirkt eine Entscheidung die Wahl zwischen $n > 2$ möglichen Folgepfaden im Graphen, so ist d um den Wert $n-1$ zu erhöhen [WM96, S. 24]. Für die Berechnung der McCabe-Metrik ist also nur die Tatsache, dass eine Verzweigung des Kontrollflusses möglich ist, von Bedeutung. Listing 2.11 zeigt die Berechnung der zyklomatischen Komplexität einer Visual Basic Funktion nach der zweiten Berechnungsmethode.

```
Public Function PrintGrade(country As String, grade As Integer) ' init d = 1
    If country = "Germany" Then          'd = 2
        Select Case grade
            Case 1                       'd = 3
                Print "sehr gut"
            Case 2                       'd = 4
                Print "gut"
            Case 3                       'd = 5
                Print "befriedigend"
            Case 4                       'd = 6
                Print "ausreichend"
            Case 5                       'd = 7
                Print "mangelhaft"
            Case 6                       'd = 8
                Print "ungenuegend"
        End Select
    ElseIf country = "Switzerland" Then 'd = 9
        Select Case grade
            Case 1                       'd = 10
                Print "ungenuegend"
            Case 2                       'd = 11
                Print "mangelhaft"
            Case 3                       'd = 12
                Print "ausreichend"
            Case 4                       'd = 13
                Print "befriedigend"
            Case 5                       'd = 14
                Print "gut"
            Case 6                       'd = 16
                Print "sehr gut"
        End Select
    End If
End Function
```

Listing 2.11: Hohe zyklomatische Komplexität trotz einfachem Programms

In diesem Programm befindet sich eine Vielzahl von Entscheidungspunkten. Der Wert von d muss in diesem Beispiel an den Statements `If`, `ElseIf`, sowie an jedem `Case` erhöht werden, da es sich hierbei um binäre Verzweigungen im Kontrollfluss handelt. Somit erreicht diese einfache Funktion eine zyklomatische Zahl von 16. McCabe zufolge sollten Programme jedoch bereits ab einem Wert von zehn als problematisch angesehen und nach Möglichkeit umgeschrieben werden, da die Fehleranfälligkeit mit einer höheren Komplexität stark ansteige.

In diesem Fall zeigt sich, dass die Zahl der Verzweigungen relativ groß ist, obwohl das Programm keinesfalls Komplex erscheint. Die zyklomatische Komplexität kann also nicht als

eindeutiger Indikator für die tatsächliche Komplexität eines Programms, im Sinne seiner Verständlichkeit, gelten. Dennoch kann sie bei der Suche nach zu komplexen und daher fehleranfälligem Code ein hilfreiches Indiz sein. Zudem gelten Programme mit einem hohen McCabe-Wert als problematisch in Bezug auf Tests und Wartung. [FP96]

2.4.4 Anforderungen an die Analyse von Visual Basic 6

Die vorgestellten Metriken definieren einen Grundstamm an Informationen, der bei der Analyse der Programme gewonnen, beziehungsweise trotz des Analysevorgangs, der von manchen Details abstrahiert, erhalten werden muss. Die folgenden Anforderungen lassen sich festhalten:

- **Betrachtung von Zeileninformationen und Kommentaren**

  Die Berechnung von Lines of Code macht es notwendig die einzelnen Zeilen des Programms auch während des Analysevorgangs nachvollziehbar zu halten. Zumindest in einer Weise, die es erlaubt, fehlende Werte aus den noch vorhandenen zu errechnen. Beispielsweise wäre $LOC - Code - Comment = Empty$. Auch die Kommentare sind zumindest bis zur Erhebung der LOC-Metriken mitzuführen, obwohl sie ansonsten keinerlei Bedeutung für den RFG haben. Alle Zeileninformationen sind für die Berechnung mindestens so lange zu erhalten, bis sie mit einer Prozedur in Verbindung gebracht werden können, da die Metriken prozedurbezogen erhoben werden.

- **Erkennen aller Binärverzweigungen**

  Die zyklomatische Komplexität kann durch die vereinfachte Formel errechnet werden, indem die Anzahl der Binärverzweigungen gezählt und um den Wert 1 erhöht wird. Dazu ist es zum einen notwendig, für Visual Basic 6 festzulegen welche Konstrukte im Quelltext als Binärverzweigung zu zählen sind und zum anderen müssen diese Strukturen bei der Analyse erkannt und bis zur Erhebung der Metrik erhalten werden.

- **Unterscheidung von Bezeichnern und Schlüsselwörtern**

  Die Halstead-Metrik erfordert eine Einteilung aller Sprachtoken in Operatoren und Operanden. Hiebei sind die Bezeichner eines Programms zweifellos als Operanden zu verstehen, während Schlüsselwörter Operatoren darstellen. Daher muss zwischen diesen beiden Wortarten unterschieden werden können, was in Visual Basic, wie ich bereits beschrieben habe, problematisch ist. Alle Token müssen im Analyseprozess so lange erhalten bleiben, bis die Unterscheidung stattgefunden hat. Erst dann kann die Halstead-Metrik berechnet werden.

Die Erhebung der Metriken wird in Abschnitt 4.3.4 beschrieben.

KAPITEL 3

Konzeption eines RFG-Schemas für Visual Basic 6

Wie bereits in Abschnitt 2.3.1 beschrieben, stellt der Resource Flow Graph ein sprachunabhängiges Modell zur Repräsentation ausgewählter Bestandteile von Programmquelltexten dar. Um ihn für die Darstellung von Programmen einer konkreten Sprache einzusetzen, ist es erforderlich, zunächst deren spezifische Elemente in Form eines Schemas auf die Konzepte des Metamodells abzubilden. Zu diesem Zweck wird in diesem Kapitel ein RFG-Schema für Visual Basic 6 Programme erstellt.

In Abschnitt 2.1 habe ich bereits die grundlegenden Bestandteile von Visual Basic 6 Programmen aufgeführt. Zwar enthält dieses Kapitel auch den Hinweis, dass die Sprache eine Vielzahl unzureichend dokumentierter Aspekte enthält, für den RFG sind jedoch, wie in Abschnitt 2.3.1 beschrieben, nur Elemente von globaler Relevanz, sowie deren Beziehungen von Interesse. Die entsprechenden Teile der Sprache sind in der vorhandenen offiziellen Dokumentation in [Mic06b] sowie in den Büchern [Mon01, Kof00, Mas99], hinreichend beschrieben. Somit ist die Definition des RFG bereits im Vorfeld weiterer Untersuchungen der Sprache möglich.

Es ist zudem notwendig den RFG vor der Analyse der Sprache zu definieren, da sich aus den darin abzubildenden Konzepten Anforderungen für die Analyse von Visual Basic 6 ergeben. Diese werden am Ende dieses Kapitels definiert. Zunächst soll jedoch ein Blick auf die bereits vorhandenen RFG-Schemata für andere Sprachen geworfen werden, da diese als Vorbild für das zu entwickelnde Schema gelten.

Kapitelinhalt

3.1 Existierende RFG-Schemata

Die Bauhaus-Dokumentation beschreibt in [Axi06] drei Schemata, die die Sprachen C, C++ und Java auf den RFG abbilden. Dort werden bereits Konzepte definiert, die auch für die Abbildung von Visual Basic 6 benötigt werden. Das neue Schema muss sich in die vorhandenen Modellierungen einpassen und bereits modellierte Konzepte wiederverwenden. Dies ist notwendig, um sicherzustellen, dass gleiche Konzepte aus unterschiedlichen Sprachen auf dieselbe Art repräsentiert werden. Dabei ist anzumerken, dass die Schemata der Sprachen C und C++ einige auch für Visual Basic 6 benötigte Konzepte einführen, diese aber mit den Bezeichnungen aus den jeweiligen Sprachen benennen, anstatt Namen zu wählen, die die Konzepte an sich auf einer abstrakten Ebene beschreiben. Im Sinne des Konsistenz werden diese auch für Visual Basic mit den vorhandenen Namen übernommen, auch wenn sie nicht zu den in der Sprache verwendeten Bezeichnungen passen. Die Umbenennung solcher Konzepte gehört nicht zu den Aufgaben dieser Arbeit.

Das RFG-Schema für C++

Das Schema für die Sprache C++ eignet sich am besten als Ausgangsbasis, weil es alle bereits bestehenden und in Visual Basic 6 zu übernehmenden Kanten- und Knotentypen enthält. Ebenso wie Visual Basic 6 erlaubt die Sprache die Verwendung sowohl rein prozeduraler, als auch objektorientierter Programmiertechniken. Die Schemata für C und Java beinhalten jeweils nur Modellierungen für eines dieser Paradigmen.

Abbildung 3.1 zeigt das Schema in einer UML-Notation. Die Erläuterungen zur Darstellungsform aus Abschnitt 2.3.1 gelten auch für die Schemata in diesem Kapitel. Es fehlen die Multiplizitäten, sowie die `Inherits` und `Overrides`-Kanten an der `Method` und die `Dispatch_Call`-Kante, die eigentlich von `Routine` zu `Method` führen müsste. Der Grund hierfür liegt darin, dass das Diagramm der Bauhaus-Dokumentation [Axi06] unverändert entnommen wurde.

Im Vergleich zum bereits vorgestellten Schema für C beinhaltet die C++-Variante zusätzlich die Konzepte, die zur Abbildung des objektorientierten Sprachanteils benötigt werden. Konkret also `Class` und `Method`. Diese sind ein weiteres Beispiel für die hierarchische Struktur der RFG-Knoten. Sie ermöglicht es in diesem Fall, Klassen bei Bedarf auch allgemein als Typen anzusehen oder beispielsweise alle Funktionen und Methoden eines Programms einheitlich als `Routine` zu betrachten. Auf die Templates werde ich nicht näher eingehen, da sie für die Abbildung von Visual Basic 6, das ein solches Konzept nicht kennt, nicht von Bedeutung sind.

Dieses Schema dient als Ausgangspunkt, von dem ich die Abbildung von Visual Basic 6, herleiten werde. Allerdings besteht eine solche Abbildung nicht nur aus einem Diagramm, das die verwendeten Konzepte darstellt, sondern auch aus Regeln, die beschreiben, welche Konzepte in welcher Art aufeinander abgebildet werden und aus den Konventionen, nach denen ein RFG erstellt wird. Hierzu gehört beispielsweise die Entscheidung, wann Elemente aus der Umgebung des Programms, beispielsweise aus Bibliotheken, aufgenommen werden.

3.2 Schema für Visual Basic 6

Ein großer Teil der in Abschnitt 2.1 vorgestellten Konzepte von Visual Basic 6 ist bereits im RFG-Modell vorhanden. In diesem Abschnitt wird untersucht inwieweit sich die Sprachelemente aus Visual Basic 6 auf die bereits vorhandenen Konzepte im RFG abbilden lassen und ob gegebenenfalls Erweiterungen des Modells vonnöten sind. Zudem gilt es festzulegen,

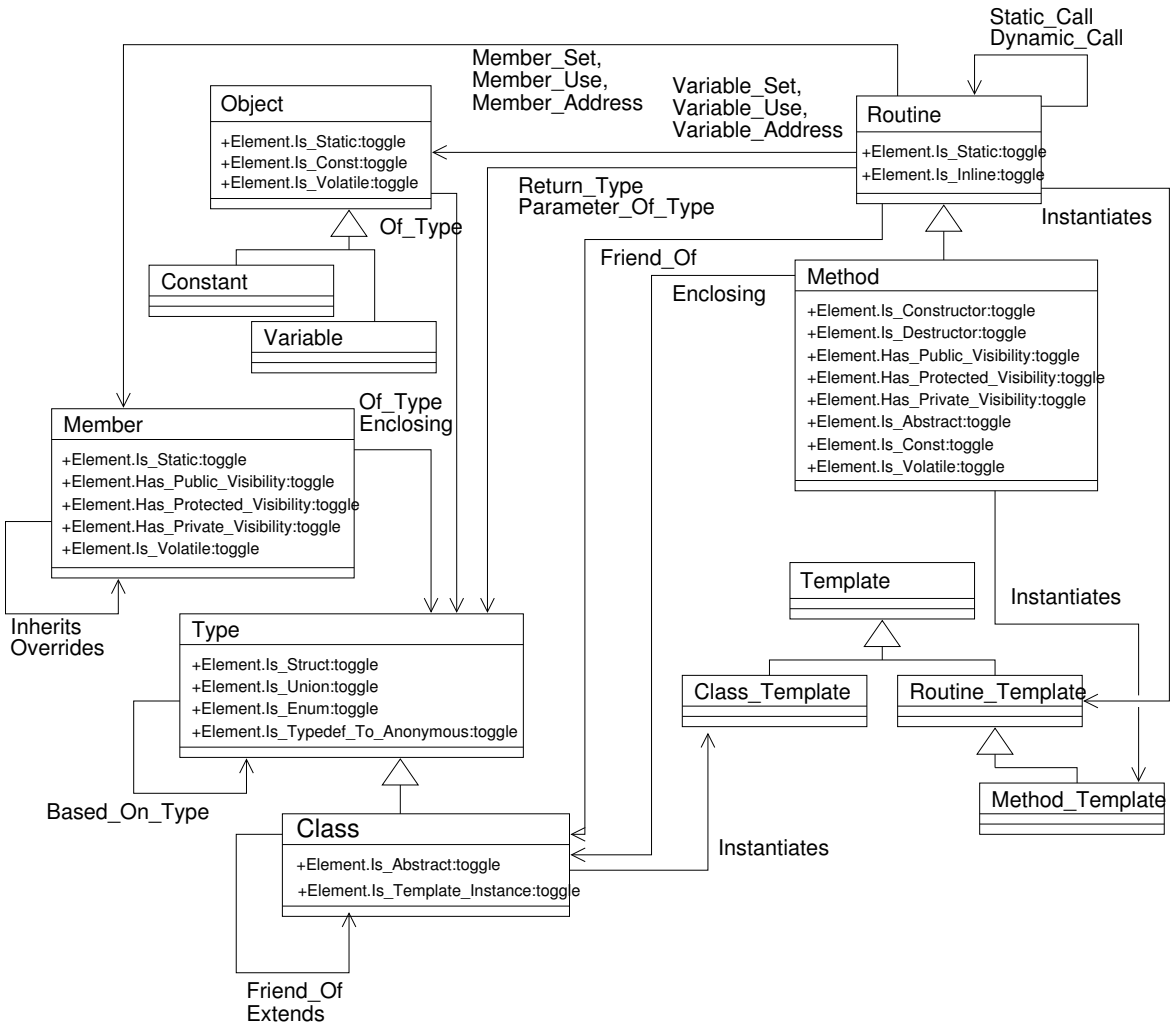

Abbildung 3.1: Das RFG-Schema für C++

welcher Anteil der Programme überhaupt relevant im Sinne des RFG ist.

Letzteres ist recht einfach festzulegen. Alle Deklarationen, die innerhalb der Prozeduren (oder Routinen im Sinne des RFG) stattfinden, sind grundsätzlich lokal. Als Entitäten können also nur die Programmelemente dienen, die auf der Modulebene definiert sind und damit global sein können. Im Folgenden sollen die noch relevanten Bestandteile von Visual Basic 6 Programmen einzeln betrachtet und die Möglichkeiten ihrer Abbildung auf den RFG diskutiert werden. Am Ende jedes Unterabschnitts werde ich alle erarbeiteten Abbildungen von Visual Basic 6 auf den RFG in Form einer Tabelle nochmals zusammenfassen. Diese gibt auch Aufschluss darüber, ob im Rahmen einer Abbildung eine Erweiterung des bestehenden Modells vorgenommen wird.

3.2.1 Module

Wie bereits beschrieben kennt Visual Basic 6 drei fundamentale Modultypen: Standard-, Klassen- und Formularmodule. Die Standardmodule dienen zur rein prozeduralen Programmierung und gruppieren Prozeduren, Variablen und Konstanten sowie Typdeklarationen in einer Datei. Demnach ist es logisch, diese Bestandteile auf den prozeduralen Anteil des RFG-Modells abzubilden. Es stellt sich jedoch die Frage, ob das Standardmodul an sich als Kon-

zept in den RFG aufzunehmen ist. Ein Blick auf die Schemata für C und C++ zeigt, dass es dort kein Konzept für ein einfaches Programmmodul, hier eine Headerdatei, auf der Ebene der Basis-Einheiten gibt. Stattdessen wird die Modulzusammengehörigkeit in den RFGs dieser Sprachen durch die Berechnung einer Modul-Sicht dargestellt. Hierbei werden die RFG-Basiseinheiten aufgrund der an ihnen annotierten Dateizugehörigkeit gruppiert und `Module`-Knoten zugeordnet, die Teil des weiteren, nicht quelltextbezogenen RFG-Metamodells sind.

Anders als in C und C++ haben die Standardmodule in Visual Basic 6 jedoch neben der einfachen Gruppierung von Elementen auch die Funktion eines Namensraumes, dessen Mitglieder über den Modulnamen qualifiziert werden können. Dies hat zur Folge, dass der RFG gleichnamige Variablen und Konstanten auf der gleichen hierarchischen Ebene enthalten kann. Abgesehen davon lässt sich ein Modul jedoch genauso verstehen, wie es bereits für C und C++ durch die Module-View umgesetzt wird. Die Art und Weise, wie die einzelnen Elemente im Quelltext referenziert werden können, stellt ein Implementierungsdetail dar, das für die Betrachtung im RFG nicht bedeutsam ist. Wichtig ist, dass die Kapselung von Basis-Einheiten in Module durch die vorhandene Implementierung der Module-View auch für Visual Basic Einheiten korrekte Darstellungen liefert, da die Konzepte sich, wie beschrieben, weitestgehend gleichen. Das Standardmodul ist daher bereits durch den `Module`-Knotentyp abgedeckt und bedarf keiner expliziten Deklaration. Es kann mit den vorhandenen Methoden zur Erstellung der Modulsicht generiert werden.

Klassen

In Bezug auf Klassen- und Formularmodule müssen zunächst die Begriffe der Klasse und des Moduls genauer betrachtet werden. In Abschnitt 2.1 habe ich bereits beschrieben, dass beide Modultypen letztendlich dazu dienen, Klassen oder Formulare, die eine Spezialisierung von Klassen sind, zu implementieren. Dies heißt jedoch nicht, dass alles, was in einem solchen Modul deklariert wird, auch in jedem Fall zu dieser Klasse gehört. Es ist ebenso möglich Aufzählungen und Typen (ein Konzept ähnlich dem `struct` aus C) in diesen Modulen zu deklarieren, wobei diese nicht der Klasse zugehörig sind, dem Modul hingegen schon. Aus der Dokumentation in [Mic06b] lässt sich ableiten, dass die in einem Klassen- oder Formularmodul deklarierten Variablen, Konstanten und Prozeduren Teil der durch das Modul repräsentierten Klasse sind, die darin deklarierten Typen hingegen nicht.

Demnach wird eine Klasse beziehungsweise ein Formular in Visual Basic 6 deklariert, indem seine Member und Methoden in einem entsprechenden Modul zusammengefasst werden. Modul und Klasse sind dabei nicht identisch, da das Modul zusätzliche Typdeklarationen enthalten kann. Für Klassen existiert bereits ein Knotentyp im RFG-Modell, für Module wie oben beschrieben auch. Zusammen erlauben beide die Funktionsweise von Modul und Klasse genau so darzustellen, wie es die Realität in Visual Basic 6 vorgibt. Die Klasse ist also ein Konzept für sich, das in einem Modul definiert wird, welches zudem auch andere Basis-Einheiten - nämlich die der im Modul deklarierten Typen - enthalten kann.

Die im Schema für C++ definierten Attribute für Klassen reichen nicht aus, um alle möglichen und für den Betrachter eventuell relevanten Informationen über Visual Basic 6 Klassen auszudrücken. Diese verfügen zusätzlich über boolesche Attribute, die ihr Verhalten hinsichtlich der Instanziierung und Sichtbarkeit von Außerhalb des Projekts steuern. 3.2.1 listet diese auf und nennt zudem die im RFG verwendeten Namen, die auf eine möglichst sprachunabhängige Bezeichnung zielen.

Das reine Hinzufügen neuer Attribute hat keinerlei negative Auswirkungen auf die sprachunabhängigen Analysen auf dem RFG. Daher ist es problemlos möglich das Modell an dieser

VB6-Name	RFG-Attributname	Beschreibung
Multi_Use	Singleton	Legt fest, ob mehrere oder nur eine Instanz eines Objekts erzeugt werden kann. Dem Singleton Entwurfsmusters nach [GHJV94] ähnlich.
Creatable	Creatable	Legt fest, ob eine Klasse direkt instanziiert werden kann, oder ob sie eine Schnittstellenmethode bereitstellen muss, die eine Instanz zur Verfügung stellt.
Exposed	Has_Public_Visibility Has_Private_Visibility	Legt fest, ob die Klasse von anderen Projekten aus sichtbar ist, sprich ob sie als Schnittstelle einer Komponente dient.
GlobalNameSpace	Global	Klassen, die dieses Attribut besitzen, machen ihre Member und Variablen in anderen Projekten, die sie einbinden, global verfügbar.

Tabelle 3.1: Abbildung von Visual Basics Klassenattributen.

Stelle zu erweitern, ohne dabei unerwünschte Seiteneffekte hervorzurufen.

Formulare

Formulare sind, wie bereits zuvor beschrieben, aus technischer Sicht Klassen mit einigen Spezialisierungen, die es ermöglichen, sie als Oberflächenkomponenten zu verwenden. Dies bedeutet, dass die Moduldateien über einen speziellen Abschnitt verfügen, in dem Visual Studio die per Drag&Drop definierten Komponenten initialisiert. Zwar wird hier eine eigene Syntax verwendet, letztendlich enthält dieser Teil aber nur Deklarationen und Zuweisungen konstanter Werte. Diese sind auf der Abstraktionsstufe des RFG jedoch nicht von Bedeutung. Letztendlich werden auch hier nur Membervariablen des Formulars deklariert und Werte zugewiesen. Also Dinge, die in Klassen gleichermaßen geschehen. Zwar werden Formulare und Klassen in der Visual Basic 6 Literatur meist getrennt voneinander beschrieben und auch in Visual Studio werden sie klar abgegrenzt, sie haben jedoch keine Attribute oder Beziehungen, die über die hinausgehen, die bereits für Klassen definiert sind.

Somit stellen sie zumindest aus der Sicht des Visual Basic Entwicklers gedanklich ein eigenes Konzept dar. Technisch sind sie dagegen bis auf wenige Sonderregeln bezüglich ihrer automatischen Initialisierung identisch mit Klassen. Auf der Betrachtungsebene des RFG sogar identisch, da eine automatische Instanziierung durch die Sprachumgebung architektonisch nicht relevant ist. Bedeutsam ist diese Eigenart nur für die Erstellung der RFG, denn hier muss die Instanzvariable hinzugefügt werden.

Eine Notwendigkeit Formulare als eigenes Konzept in den RFG einzuführen, um Visual Basic 6 Programme darstellen zu können, besteht demnach nicht. Zudem ist fraglich, ob es dem Streben nach Sprachunabhängigkeit nicht eher abträglich ist, das Formular als neues Konzept einzuführen. Andere Sprachen wie C++ oder Java ermöglichen ebenfalls die Programmierung von Formularen, verwenden dazu aber in der Regel Klassen als Grundlage (siehe Java Swing[1] oder Motif[2]). Es ist möglich, die Visual Basic Formulare auf Klassen abzubilden. Dies würde

[1] `http://java.sun.com/javase/technologies/desktop/`
[2] `http://www.opengroup.org/motif/`

jedoch den Verlust der Information, dass es sich um ein Formularmodul handelt, bedeuten. Im Sinne des Anwenders wäre es wünschenswert, dass Formulare auch als solche im Graphen zu erkennen sind – gerade deswegen, weil sie ein wichtiges Programmelement in Visual Basic darstellen.

Einen Kompromiss bietet die Möglichkeit, Attribute in Gravis zu visualisieren. Dies wird beispielsweise verwendet, um `structs` in C oder C++ als solche erkennbar zu machen. Ein `struct` wird durch den Knotentyp `Type` abgebildet, der auch zur Modellierung von anderen Typarten dient. Somit wird der `struct` im Graphen durch ein allgemeines Konzept repräsentiert. Seine tatsächliche Form wird mit booleschen Attributen an den `Type`-Knoten annotiert. Das Vorhandensein eines solchen Attributes wird mit einem kleinen Zusatzbild kenntlich gemacht, dass dem Typ-Symbol angehängt wird. Damit bleibt das konzeptionelle Modell rein von sprachspezifischen Details, die dennoch für den Betrachter eindeutig erkennbar sind. Für Formulare lässt sich diese Lösung adaptieren, indem der Knotentyp `Class` um das boolesche Attribut `Form` erweitert wird.

Abbildung 3.2: Attributvisualisierung in Gravis. Das S weist den Typen als Struct aus.

Zusammenfassend lassen sich die Module in Visual Basic korrekt durch das bereits im RFG vorhandene Konzept `Module` abbilden. Klassen und Formularen werden auf das bestehende `Class` abgebildet, das um vier boolesche Attribute erweitert wird.

Die Attribute `Has_Public_Visibility` und `Has_Private_Visibility` sind zwar im C++ Schema für Klassen nicht vorgesehen, allerdings werden sie in den RFGs für Java-Programme bereits genau in diesem Sinne genutzt.

Beziehungen von Klassen und Formularen

In der Modellierung für C++ stellt die Kante `Extends` von einer Klasse zu einer anderen die Generalisierung dar. Ich habe zuvor beschrieben, dass Visual Basic nur über eine eingeschränkte Objektorientierung verfügt. Es ist dem Programmierer nicht möglich, Vererbungshierarchien zu definieren. Lediglich Interfaces lassen sich implementieren. Eine Vererbung von Funktionalität oder Daten steht nicht als Konzept zur Verfügung. Bei einer genaueren Betrachtung zeigt sich allerdings, dass Vererbung in Visual Basic durchaus ein Konzept ist, auch wenn es dem Programmierer nicht zur Verfügung steht. Konkret zeigt sich das an den sprachzugehörigen Klassen `Form` und `Control`, die die Basis für benutzerdefinierte Formulare und Steuerelemente darstellen. Jedes konkrete Formular verfügt stets über Standardmethoden und -eigenschaften wie beispielsweise eine Breite `Width` oder diverse Events und zwar auch dann, wenn diese nicht vom Benutzer definiert sind. Offensichtlich findet hier implizit Vererbung statt. Da der RFG bereits ein Konzept hierfür beinhaltet, spricht nichts dagegen, dieses auch für Visual Basic 6 einzusetzen.

Die in Abschnitt 2.1.2.2 beschriebene Interface-Implementierung kann nicht durch die Verwendung des `Extends`-Kantentyps abgebildet werden, da es sich dabei nicht um Vererbung

handelt. Das RFG-Schema für Java kennt einen Kantentypen `Implementation_Of`, der dort Klassen und Interfaces verbindet und die Implementierung des Interfaces repräsentiert. Ich übernehme diese Kante in das Schema für Visual Basic 6, um die Implementierung eines Klasseninterfaces durch eine andere Klasse zu modellieren. Die Kante ist in einem konkreten Modell von der implementierenden Klasse auf die interfacedefinierende Klasse gerichtet.

Weitere Beziehungen werde ich diskutieren, sobald die anderen Knotentypen, die daran teilnehmen, für den Visual Basic 6 definiert sind.

Visual Basic 6 →	RFG	Neu
Moduldatei	`Module`-Knoten in Modul-Sicht	
Klasse	`Class`-Knoten	
Formular	`Class`-Knoten mit `Form`-Attribut	×
Klassenattribut GlobalNameSpace	`Global`-Attribut	×
Klassenattribut Creatable	`Singleton`-Attribut	×
Klassenattribut Exposed	Attribute `Has_Public_Visibility` und `Has_Private_Visibility`	
Klassenvererbung	`Extends`-Kante	
Interface-Implementierung	`Implementation_Of`-Kante	

3.2.2 Typen

Analog zur vorhandenen Modellierung sind Klassen in Visual Basic 6 als eine Generalisierung von Typen zu verstehen. Weitere Typarten sind die Enumeration und eine komplexe Form, die dem `struct` aus C ähnelt. Es handelt sich dabei um einen Verbund von Variablen beliebiger Datentypen, der mit dem Schlüsselwort `Type` deklariert wird. Um Unklarheiten und Verwechslungen mit dem allgemeinen Konzept des Typen oder dem RFG-Knotentyp `Type` zu vermeiden, werde ich für das Visual Basic Konstrukt fortan den Namen VB6-`Type` verwenden.

Enumerationen sowie `structs` sind im RFG bereits über den `Type`-Knotentyp und seine Attribute modelliert. Zwar trifft die Bezeichnung `struct` für Visual Basic nicht zu, das damit ausgedrückte Konzept ist jedoch das gleiche. Daher werden VB6-`Type`s als Typen mit `struct`-Attribut dargestellt. Enumerationen (in Visual Basic verwenden diese wie in C das Schlüsselwort `Enum`) lassen sich analog zu ihren C-Pendants abbilden. Primitive Typen sind nicht weiter zu betrachten, da sie nicht im RFG abgebildet werden.

Visual Basic 6 →	RFG	Neu
`Enum`	`Type`-Knoten mit `Enum`-Attribut	
`Type`	`Type`-Knoten mit `Struct`-Attribut	

3.2.3 Variablen und Konstanten

Variablen und auch Konstanten lassen sich in Visual Basic 6 in einem prozeduralen sowie als Member im objektorientierten Kontext verwenden. Ihre Abbildung auf die Knotentypen `Variable`, `Constant` und `Member` ist einleuchtend. Ebenso ihre `Of_Type`-Beziehung zum `Type`-Knotentyp. Da ich zudem auch das Konzept der Vererbung in das Schema für Visual Basic übernehme, ist es konsequent, auch die `Inherits`- und `Overrides`-Kantentypen zu übernehmen. Ebenso ist die `Enclosing`-Kante von einem `Member` hin zu einem Type zu übernehmen, denn in Visual Basic kann neben Klassen auch der VB6-`Type` Member enthalten. Aufzählungskonstanten, die in `Enum`s deklariert werden, bildet der RFG für C++ als `Constant` ab. Auch dies lässt sich direkt übernehmen.

Die Modellierung für diese Entitäten erfolgt also analog zum C++-Schema. Lediglich die Attribute `Volatile` und `Const` sowie `Static` für Member sind nicht notwendig, da es in Visual Basic 6 keine entsprechenden Konzepte gibt. Das `Static`-Attribut des `Object`-Knotens wird übernommen und mit der gleichen Semantik wie in C verwendet, indem es die Sichtbarkeit einer Variablen oder Konstante abbildet, die in Visual Basic durch die Schlüsselwörter `Public` und `Private` gesteuert wird.

Visual Basic 6 →	**RFG**	**Neu**
Variable auf globaler Ebene eines Standardmoduls	`Variable`-Knoten	
Konstante auf globaler Ebene eines Standardmoduls	`Variable`-Knoten	
Variablen- & Konstantensichtbarkeit in Standardmodulen	`Static`-Attribut	
Variable oder Konstante in Klasse oder Formular	`Member`-Knoten	
Typ von Variablen, Konstanten oder Membern	`Of_Type`-Kante	
Vererbung von Membern	`Inherits`-Kante	
Zugehörigkeit von VB6-`Type`-Membern	`Enclosing`-Kante	

3.2.4 Routinen und Methoden

Als Routinen und Methoden sollen bewusst zunächst einmal nur die gewöhnlichen Varianten betrachtet werden, die in Visual Basic 6 durch die Schlüsselwörter `Sub` (Prozedur ohne Rückgabewert) und `Function` deklariert werden. Die in Abschnitt 2.1 beschriebenen Propertys und Events, werden später separat betrachtet, da diese tatsächlich Konzepte darstellen, die sich nicht direkt auf das vorhandene RFG-Modell abbilden lassen. Für die gewöhnlichen Prozeduren ist das jedoch möglich. Je nach dem Ort ihrer Deklaration, können sie auf die Konzepte `Routine` und `Method` abgebildet werden.

Das `Is_Static`-Attribut wird dabei analog zu dem zuvor übernommenen, gleichnamigen Attribut von `Object` auch in das neue Schema übernommen. Für das `Is_Inline`-Attribut gibt es dagegen keine Verwendung. Dafür kennt Visual Basic jedoch ein Konzept, das bislang nicht im RFG-Modell bekannt ist. Bedauerlicherweise trägt es in der Sprache ebenfalls den Namen `Static`, hat aber eine vollkommen andere Semantik als das vorhandene `Is_Static`-Attribut des RFG. Eine in Visual Basic 6 mit dem Modifizierer `Static` versehene Prozedur macht alle in ihr deklarierten Variablen statisch, so dass sie ihren Wert über den Aufruf hinaus behalten. Dies gilt auch für statische Methoden, die nicht, wie in anderen objektorientierten Sprachen üblich, ohne Instanz aufgerufen werden können. Dies wird der Modellierung in Form eines neuen `Is_Persisting`-Attributs hinzugefügt. Der Name soll auf die datenerhaltende - also *persistente* - Eigenschaft Bezug nehmen.

Klassen verfügen in Visual Basic über spezielle Methoden, die Konstruktoren und Destruktoren ähneln, da sie automatisch zu Beginn, beziehungsweise Ende, der Lebensdauer einer Klasse oder eines Formulars aufgerufen werden. Diese Funktionen tragen stets den Namen `Class_Initilize` oder `Form_Initialize` beziehungsweise `Class_Terminate` oder `Form_Terminate`. Diese werden als Konstruktoren und Destruktoren verstanden und mit den entsprechenden Attributen abgebildet. Die Sichtbarkeit einer Methode kann die Werte `Private` und `Public` annehmen, alle weiteren Attribute des C++-Schemas treffen nicht auf Visual Basic zu.

Visual Basic 6 →	RFG	Neu
`Sub` oder `Function` in Standardmodul	`Routine`-Knoten	
`Sub` oder `Function` in Klasse oder Formular	`Method`-Knoten	
Sichtbarkeit einer `Sub` oder `Function` in Standardmodul	`Is_Static`-Attribut	
Static-Attribut einer `Sub` oder `Function` in Standardmodul	`Is_Persisting`-Attribut	
`Class_Initialize`, `Class_Terminate` und entsprechende Formularmethoden	`Is_Constructor`- und `Is_Destructor`-Attribut	
Sichtbarkeit von Methoden	`Has_<...>_Visibility`-Attribute	

Kanten

Die Kantentypen `Return_Type` und `Parameter_Of_Type` lassen sich direkt übernehmen. Zu den `Use`- und `Set`-Kantentypen ist anzumerken, dass `Set` in Visual Basic als die Operation verstanden wird, die einer Variable eine *Objektreferenz* zuweist. Die Zuweisung eines *Wertes* nennt sich dagegen `Let`. Eine `Let`-Operation auf einer Objektvariable setzt die *Standardeigenschaft* des Objektes, also ein Attribut und nicht die Referenz. Analog zu der Modellierung von C++ und Java werden beide Operationen als `Variable_Set` beziehungsweise `Member_Set` abgebildet. Zeigt eine solche Kante auf eine Objektvariable, so steht sie für das Setzen der Objektreferenz. Wird dagegen eine Standardeigenschaft gesetzt, so wird eine entsprechende Kante zum Knoten dieser als Standard festgelegten Eigenschaft gezogen.

Den Aufruf von Routinen und Methoden modelliert der RFG mit den drei Konzepten `Static_Call`, `Dynamic_Call` und `Dispatch_Call`, die verschiedene Formen des Bindens darstellen. Statisch ist hierbei das frühe Binden, bei dem zum Übersetzungszeitpunkt klar ist, welcher Programmcode durch einen im Quelltext enthaltenen Aufruf ausgeführt werden soll. Alle in Standardmodulen enthaltenen Prozeduren werden nach diesem Prinzip aufgerufen. Der dynamische Aufruf bezieht sich in der Modellierung von C und C++ auf den Aufruf einer Routine über einen Funktionszeiger. Hierbei handelt es sich um spätes Binden, da der auszuführende Programmcode sich aus dem laufzeitabhängigen Funktionszeiger ergibt. Einen solchen dynamischen Aufruf von prozeduralem Programmcode kennt Visual Basic 6 jedoch ebensowenig wie Zeiger im Allgemeinen. Die dritte Variante, der `Dispatch_Call`, modelliert Aufrufe von polymorphen Funktionen in einer Vererbungshierarchie. In C++ findet der Aufruf als `virtual` deklarierter Methoden immer über eine virtuelle Tabelle statt, die zur Laufzeit Aufschluss über die aufzurufende Implementierung der Methode gibt. Nicht als `virtual` definierte Methoden werden dagegen statisch aufgerufen.

Auch Visual Basic 6 kennt das Prinzip der Polymorphie. Aus [Mic06b, Mic03] geht hervor, dass hier ebenfalls eine virtuelle Tabelle, die *vtable*, dazu verwendet wird. Im Gegensatz zu C++ bietet die Sprache aber kein Schlüsselwort, das dieses Verhalten ein- oder abschaltet. Daher kann jede Methode potentiell per Dispatching aufgerufen werden. Wann welches Bindungsverhalten zum Einsatz kommt, lässt sich anhand des P-Codes überprüfen. Der *VB Decompiler 3.0*[3] kann P-Code disassemblieren und einige der Opcodes namentlich aufführen. Hierbei fällt auf, dass alle Aufrufe von Methoden – unabhängig davon, ob sie von einer anderen Klasse nochmals implementiert werden – stets einen `VCall`, was auf die vtable hindeutet, ausführen, während bei Aufrufen von Routinen immer direkte Sprungpositionen angegeben werden. Diese Beobachtung passt zu der Aussage von [Mon01, S. 144], nach der kein eigenes Compiler-

[3] VB Decompiler: `http://www.vb-decompiler.org/`

Backend für Visual Basic 6 existiert und stattedessen intern das Backend von Visual C++ eingesetzt wird, um VB6-Programme zu erzeugen. Nähere Informationen über das Bindungsverhalten von Visual Basic 6 konnten nicht gefunden werden, doch aus den Beobachtungen am P-Code und den weiteren Informationen der Dokumentation lässt sich vermuten, dass der Aufruf von Methoden generell anhand des `Dispatch_Call`s erfolgt. Zumindest konnte kein Fall herbeigeführt werden, in dem der Aufruf einer Methode durch frühes Binden erfolgte.

Das bereits angesprochene Fehlen von Zeigern und die strikt umgesetzte Abstraktion von Speicheradressen bedeutet, dass es keine Verwendung für die Kantentypen `Variable_Address` und `Member_Address` gibt. Ebenso kennt Visual Basic in Version 6 keine Templates, womit `Instantiates` ebenfalls nicht von Bedeutung ist. Allerdings gibt es ein Konzept, das sich `Friend` nennt. Bedauerlicherweise hat dieses wiederum eine leicht andere Bedeutung als beispielsweise in C++. Dort erlaubt es Methoden oder ganzen Klassen auf die eigentlich privaten Member und Methoden anderer Klassen zuzugreifen. In Visual Basic 6 ist es eine Form der Sichtbarkeit und stellt eine Alternative zu den Schlüsselwörtern `Public` und `Private` dar. Eine als Friend deklarierte Methode ist von allen anderen Modulen eines Projekts aus sichtbar, jedoch nicht von außerhalb des Projekts. Sie ist also nicht Teil einer Schnittstelle, selbst wenn sie einer COM-Schnittstellen-Klasse angehört. Dies ist also ein anderes Konzept als jenes, das die `Friend_Of`-Kante modelliert. Zur Modellierung dieser speziellen Sichtbarkeitsform wird das Attribut `Has_Friend_Visibilty` eingeführt. Die existierende Kante kommt nicht zur Verwendung.

Visual Basic 6 →	**RFG**	**Neu**
Aufruf einer Prozedur eines Standardmoduls	`Static_Call`-Kante	
Aufruf einer Methode	`Dispatch_Call`-Kante	
Zugehörigkeit einer Methode zu einer Klasse bzw. einem Formular	`Enclosing`-Kante	
Typ des Rückgabewerts einer Prozedur	`Return_Type`-Kante	
Typ eines Parameters einer Prozedur	`Parameter_Of_Type`-Kante	
Setzen eines Wertes	`Variable_Set`- oder `Member_Set`-Kante zur Variable bzw. Member	
Setzen einer Objektreferenz	`Variable_Set`- oder `Member_Set`-Kante zur Variable bzw. Member	
Lesen eines Wertes	`Variable_Use`- oder `Member_Use`-Kante zur Variable bzw. Member	

3.2.5 Propertys

Propertys sind, wie bereits in Abschnitt 2.1 beschrieben, ein besonderer Fall. Betrachtet man eine Property von außen, so wirkt sie wie eine einfache Variable und wird auch so verwendet. Im Inneren ist sie jedoch eine Gruppe von Property-Prozeduren, die zusammen das von außen sichtbare Symbol deklarieren. Das RFG-Modell enthält bereits Konzepte für Variablen und Member beziehungsweise Routinen und Methoden. Jedoch verfügt es über nichts, das etwas modelliert, das beide Gestalten annimmt. Zunächst muss ermittelt werden, welche Informationen bezüglich einer Property zu modellieren sind. Dann können mögliche Modellierungen erarbeitet und bewertet werden.

Zugriffs-Prozeduren

Ich betrachte zunächst die Zugriffs-Prozeduren der Propertys. Die Implementierung dieser Prozeduren erfolgt auf gleiche Weise wie die von gewöhnlichen Prozeduren. Demnach können darin Variablen- und Memberverwendungen sowie Routinen- und Methodenaufrufe enthalten sein. Die Verwendung der Property führt implizit zum Aufruf einer ihrer Zugriffs-Prozeduren, wobei die Art der Verwendung entscheidend dafür ist, welche Prozedur aufgerufen wird. Alle diese Informationen sind für den RFG relevant und werden für Routinen in Form von Kanten modelliert. Sie müssen auch für die einzelnen Property-Prozeduren modelliert werden, da die Analysen auf dem RFG diese Beziehungen verwenden. Die Untersuchung eines Call-Graphen kann nur dann vollständig durchgeführt werden, wenn auch die Aufrufe von Zugriffsprozeduren, sowie die Aufrufe, die sie selbst tätigen, auch abgebildet werden. „Lücken“ im Call-Graphen sollten vermieden werden.

Die Property-Prozeduren sind Routinen also sehr ähnlich. Sie unterscheiden sich jedoch durch die Tatsache, dass es mehrere gleichnamige Prozeduren gibt, die mit `Get`, `Let` und `Set` unterschiedliche Zugriffe definieren. Damit sie bei der Betrachtung des RFG unterschieden werden kann, muss diese Information ebenfalls abgedeckt werden. Auf den ersten Blick bietet es sich also an, sie als Spezialisierung von Routinen zu modellieren. Dieser Versuch scheitert jedoch, da Propertys sowohl in Standardmodulen, als auch in Klassen vorkommen können. Der bisherigen Modellierung zufolge könnten sie also je nach Ort ihrer Deklaration eine Spezialisierung von `Routine` oder `Method` darstellen. Es würde nicht reichen, sie nur als Spezialisierung von `Routine` zu definieren, da `Method` diese Spezialisierung nicht erben würde. Die Einführung von zwei Spezialisierungen und damit zwei Knotentypen für die Zugriffs-Prozeduren wäre redundant und würde das RFG-Modell aufblähen. Abbildung 3.3 stellt diese Modellierungsversuche dar.

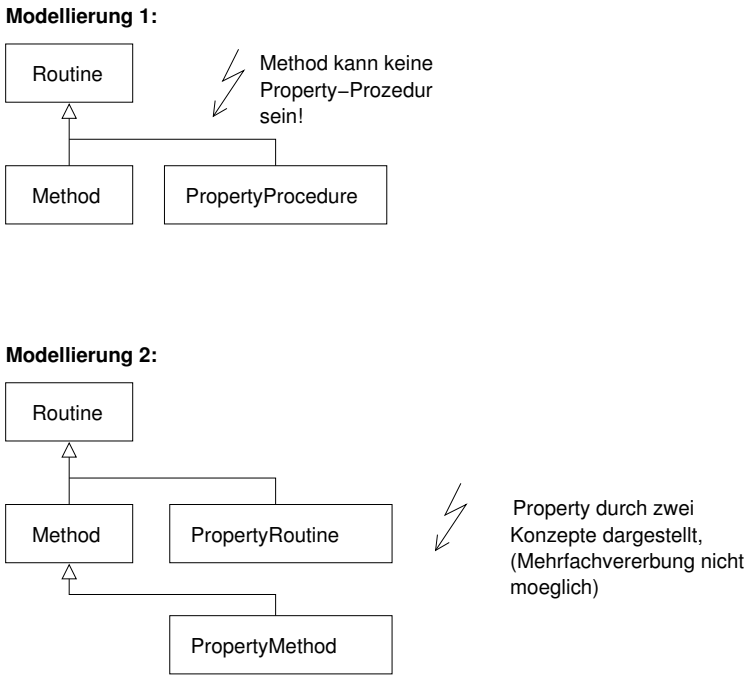

Abbildung 3.3: Problematische Modellierungen für die Property-Prozedur

Eine Alternative besteht in der Art, die zur Modellierung von Formularen verwendet wurde, sprich in einem visualisierbaren booleschen Attribut. Dieses kann der `Routine` hinzugefügt werden und würde damit auch von `Method` geerbt werden. Als Name für das Attribut bietet

sich `Property_Accessor` an. Die Existenz der Zugriffs-Prozeduren und ihrer mit der normalen Prozedur identische Semantik wären damit abgebildet. Es fehlt jedoch noch die Information, welchen Zugriffstyp eine solche Prozedur behandelt. In dieser Modellierung bieten sich zwei Wege zur Darstellung der Zugriffsart an:

1. Durch eine Zerlegung des `Property_Accessor` Attributes in drei einzelne Attribute `Property_Accessor_Get`, `Property_Accessor_Let` und `Property_Accessor_Set`

2. Durch die Einführung einer Namenskonvention, die den für alle Zugriffs-Prozeduren gleichen Namen um ein Suffix erweitert, das deren Zugriffsart definiert. Als Suffix können <get>, <let> und <set> verwendet werden.

Ansatz 1 führt drei Attribute für letztendlich nur ein Konzept ein. Der zweite Ansatz bietet eine schlankere Modellierung. Da die Art des Zugriffs im Wesentlichen von informativem Charakter ist, habe ich den letzten Ansatz gewählt. Das Konzept der Zugriffs-Prozedur ist damit abgebildet und die Knoten des RFG bleiben für den Betrachter einfach als Property zu erkennen. Der Zugriffstyp ist ebenso ersichtlich.

Das Property-Symbol

Betrachtet man Propertys von außen, so erscheinen sie wie eine Variable oder ein Member. Der Einfachheit halber nenne ich diese Betrachtungsweise einer Property das *Property-Symbol*. Für den Visual Basic Programmierer dürfte diese Sichtweise die in der Praxis am häufigsten verwendete Wahrnehmung von Propertys sein. Daher könnte diese Information im RFG, der das Programmverstehen unterstützen soll, nützlich sein. Zudem verfügt der RFG bereits über Konzepte, die die Verwendung von Datenstrukturen modellieren. Es muss geprüft werden, inwiefern sich ein solches Konzept einfügen lässt.

Das RFG-Metaodell kennt die Knotentypen `Variable` und `Member`. Hier stellt sich das gleiche Problem wie zuvor bei den Zugriffs-Prozeduren, da Propertys je nach Ort ihrer Deklaration das eine oder das andere darstellen können. Im Gegensatz zu `Routine` und `Method` gibt es in der Vererbungshierarchie jedoch keine direkte Beziehung zwischen `Variable` und `Member`. Beide Konzepte liegen orthogonal zueinander in der Hierarchie. Ein neuer Knotentyp scheidet damit bereits aus, sofern dieser nicht ebenfalls orthogonal zu `Variable` und `Member` deklariert würde. Dies entspräche aber nicht seiner Semantik. Es bleibt jedoch die Möglichkeit, Property-Symbole selbst als schlichte Variablen oder Member zu verstehen und einen Knoten des entsprechenden Typs zu definieren. Dieser könnte wiederum mit entsprechenden Use- und Set-Kanten mit den aufrufenden Routinen verbunden werden.

Diese Art der Abbildung ist jedoch redundant, denn gäbe es sowohl die Zugriffs-Prozeduren als auch das Symbol, so müsste es für eine Property-Verwendung auch zwei Kanten geben, die den gleichen semantischen Hintergrund repräsentieren. Ein viel größeres Problem liegt jedoch darin, dass das Property-Symbol mit einer Use- oder Set-Kante versehen werden müsste. Diese Kanten stellen jedoch Datenabhängigkeiten im Programm dar. Wie ich in Abschnitt 2.1.2.2 beschrieben habe, ist die Implementierung der Prozeduren allein dem Programmierer überlassen. Es besteht also nicht zwangsläufig eine Datenabhängigkeit beim Aufruf einer solchen Prozedur. Der Schluss, dass die Verwendung einer Property zu einer Use- oder Set-Kante führt, ist daher gar nicht zulässig. Die Abbildung des Property-Symbols als `Variable` oder `Member`, der über keine eingehenden Kanten verfügt, wäre allerdings wenig sachdienlich. Daher stellt diese Modellierung keine sinnvolle Option dar.

Fazit

Es erweist sich als sinnvoller, die Sichtweise einer Property als einzelnes Symbol statt als Sprachdetail zu verstehen, das keiner Abbildung im RFG bedarf. Tatsächlich ist es so, dass eine Property-Prozedur *beliebige* Anweisungen enthalten und somit durchaus zu einem reinen Unterprogramm degenerieren kann. Somit ist der Effekt einer Propertyverwendung primär der Aufruf einer Prozedur, nicht zwingend aber eine Verwendung von Datenstrukturen. Da die Abbildung der Prozeduren unverzichtbar ist, die des Property-Symbols jedoch primär von informativem Charakter, beschränkt sich die gewählte Modellierung auf die Prozeduren und die Aufrufsemantik. Das Modell erlaubt zudem die Zusammengehörigkeit von Property-Prozeduren anhand ihrer Attribute und Namen aus dem RFG zu derivieren. Diese Information ist also nicht verloren, sondern implizit im Graphen vorhanden.

Die gewählte Modellierung erfordert lediglich ein weiteres Attribut sowie eine Namenskonvention und hat somit einen minimalen Effekt auf das bestehende Modell.

Visual Basic 6 →	**RFG**	**Neu**
Property-Prozedur	`Routine` bzw. `Member` mit `Is_Property_Accessor`-Attribut	×
Zugriffsart einer Property-Prozedur	Suffix <get>, <let> oder <set> am Kontennamen	×

3.2.6 Events

Events stellen eine wirkliche Neuerung für das RFG-Modell dar, da es ein solches Konzept bislang nicht gibt. Zwar werden in den bereits abgebildeten Sprachen durchaus auch Ereignisse verwendet, beispielsweise im bereits erwähnten Java Swing, jedoch geschieht dies hier über Objekte, nicht über gesonderte spracheigene Konzepte. Aus 2.1.2.2 sind die Bestandteile, die zum eventbasierten Programmablauf verwendet werden, bekannt. Diese sind:

- Die Eventdeklaration selbst.
- Eine Variable von Typ der Klasse, die das Event definiert.[4]
- Eine `Sub`-Prozedur, die das Event für diese Variable behandelt.
- Eine Prozedur, die das Event auslöst.

Events sind ein wesentlicher Bestandteil des Programmablaufs in Visual Basic 6 und schon daher auch im Sinne des RFG von Interesse. Aus der Beschreibung der Semantik eines Events in 2.1.2.2 geht hervor, dass es von Prozeduren ausgelöst wird und selbst Prozeduraufrufe zur Folge hat. Das bedeutet, dass es Bestandteil der Call-View eines RFG sein sollte und entsprechend zu modellieren ist. Den Ausführungen über die Modellierung von Aufrufkanten aus dem vorhergehenden Teil folgend, sollte die Aufrufsemantik von Events also durch Kanten modelliert werden, die sich von dem abstrakten `Call`-Kantentyp ableiten. Dazu kann bereits jetzt festgehalten werden, dass das Auslösen eines Events, das ich `Raise_Event` nenne, von einer Methode ausgehen muss (Events können nur in Klassen und Formularen definiert und nur dort auch ausgelöst werden) und dass der aus der Eventauslösung resultierende Aufruf `Routine` als Ziel hat, denn Events lassen sich sowohl in Klassen, als auch in Standardmodulen behandeln.

[4] Zudem wird eine Instanz der Klasse benötigt. Diese ist im Sinne des RFG aber nicht relevant

Bevor diese Kanten näher definiert werden können, müssen jedoch das Event selbst und die eventbehandelnde Prozedur modelliert werden. Letzteres ist eine einfache Aufgabe und analog zu den Propertys zu lösen. Die Eventbehandlung erfolgt in `Sub`-Prozeduren, die nur durch die Zusammensetzung ihres Namens zur eventbehandelnden Prozedur werden. Ihr direkter Aufruf ist nach wie vor möglich. Im RFG können sie also problemlos als `Routine` oder `Method` dargestellt werden. Ein neues Attribut namens `Event_Sink` ermöglicht es, die Tatsache, dass die Prozedur ein Event behandelt, zu visualisieren. Da eine solche Prozedur stets ein bestimmtes Event an einer bestimmten Variable behandelt, ist auch diese Information abbildbar. Hierzu wird die neue Kante `Handles_Event_Of` eingeführt, die von einer Routine auf `Objekt` und `Member` weisen kann.

Weniger einfach ist die Modellierung des Events selbst. Einiges spricht dafür, dass es einer Prozedur ähnlich ist. Es hat eine ähnliche Signatur, kann über Parameter verfügen, ruft selbst andere Prozeduren auf und wird in der Literatur gelegentlich auch als „Ereignisprozedur“ bezeichnet. Allerdings fehlt ihm ein Rumpf, der eine Implementierung enthalten kann. Seine Auslösung erfolgt über ein gesondertes Statement und die von ihm getätigten Aufrufe unterliegen besonderen Regeln. Zudem treffen die für `Routine` und `Method` vorgesehenen Attribute nicht auf es zu. Eine Modellierung über ein visualisierbares Attribut würde das Event zwar sichtbar machen, allerdings bietet dieses Mittel keine sichtbare Abhebung wie es der Bedeutung des Konzeptes angemessen wäre und wird in der Modellierung zudem bereits mehrfach eingesetzt.

Abgesehen davon ist nicht zu erwarten, dass die Modellierung eines Ereignis-Konzeptes in Form einer Routine eine in Programmiersprachen allgemein übliche Gegebenheit wiederspiegelt. Die Darstellung des Events als eigenes Konzept, das von `Routine` und `Method` losgelöst ist, stellt dagegen eine generische Modellierung dar. Daher habe ich diese gewählt.

Eventauslösung

Die benötigten Kanten und Knoten zur Modellierung des Events sind damit festgelegt. Events können in Methoden aufgerufen werden, weshalb eine `Raise_Event`-Kante von `Method` hin zu `Event` benötigt wird. Die in Folge einer Eventauslösung aufgerufene Routine muss mit einer eingehenden `Handled_By`-Kante versehen werden, die ihren Ursprung am `Event` hat. Ich habe ebenfalls beschrieben, dass Events zusammen mit ihrer Aufrufsemantik Teil des Call-Graphs sein sollten, da Routinen aufgerufen werden. Es liegt also nahe, dass die beiden neuen Kanten `Raises_Event` und `Handled_By` in der Kantenhierarchie des RFG vom Konzept `Call` abzuleiten sind. Somit ergibt sich für die Eventmodellierung das in Abbildung 3.4 dargestellte Teilschema.

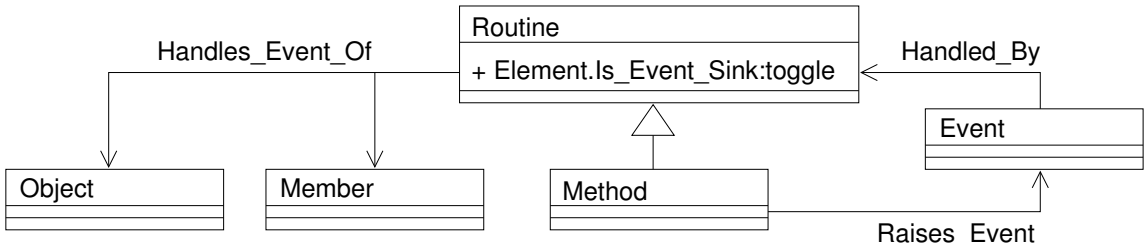

Abbildung 3.4: Teilschema für die Event-Modellierung

Zu beachten ist hierbei die gewählte Kantenrichtung, die von wesentlicher Bedeutung für eine Betrachtung der Aufrufstrukturen ist. Die `Handled_By`-Kante, die ein Event mit einer behandelnden Routine verbindet, ist auf letztere gerichtet, da dies der Aufrufrichtung entspricht.

Somit ist es weiterhin möglich, Aufrufe zu verfolgen, indem man den `Call`-Kanten folgt.

Visual Basic 6 →	RFG	Neu
Event	`Event`-Knoten	×
Eventauslösung	`RaisesEvent`-Kante von `Method` zu `Event`	×
Behandlung eines Events durch Prozedur	`Event_sink`-Attribut an `Routine` bzw. Method und `Handled_By` Kante vom `Event` zur `Routine` oder `Method`	×
Behandlung eines Events einer Variable	`Handles_Event_Of`-Kante von behandelnder `Routine` bzw. `Method` zur `Variable` bzw. zum `Member`	×

Das Gravis-Icon für Events ist ein gelbes gleichseitiges Dreieck, das nach rechts zeigt und den Anstoß eines Prozesses aus dem wartenden Zustand symbolisieren soll. Es ist in Abbildung 3.5 dargestellt.

Abbildung 3.5: Gravis-Icon für neuen Event-Knoten

3.2.7 Unbekannte Symbole

In Abschnitt 2.2.1 habe ich die Problematik beschrieben, die aus der besonderen Form des späten Bindens im Component Object Model erwächst. In einigen Fällen ist es demach nicht möglich, anhand statischer Analysen zu bestimmen, welcher Methode, beziehungsweise welchem Attribut, ein Bezeichner im Quelltext zuzuordnen ist.

Wie ich in Abschnitt 2.1.3 beschrieben habe, ist die Bedeutung von Bezeichnern im Quelltext nicht aus der Syntax ersichtlich, da diese mit ihren vielen unterschiedlichen Konventionen für Klammerungen keine eindeutige Unterscheidung erlaubt. Dieses Problem lässt sich nur durch die Betrachtung der Semantik und vollständigen Informationen über die Symbole des Programms lösen. Wenn bekannt ist, welches Symbol ein Bezeichner referenziert, so kann auch ermittelt werden, was für eine Art der Referenzierung stattfindet. Allerdings fehlt diese Information bei den spät gebundenen Bezeichnern, so dass nur die Syntax als Anhaltspunkt bliebt.

Wüsste man, was für eine Art von Symbol referenziert wird, so könnte man einen Knoten mit dem entsprechenden Typ und dem Namen des Bezeichners erstellen, zu dem man eine passende Kante zieht. Damit wäre zwar nicht die Frage nach der Identität des aufgerufenen Objektes geklärt, aber immerhin klar, von welcher Art es ist und wie es aufgerufen wird. Aber die Syntax gibt diese Information nicht her. Es scheitert schon daran, dass ein Bezeichner ohne Klammerung prinzipiell auf eine Variable oder auf eine Property verweisen kann. Beides sind unterschiedliche Konzepte, die auf unterschiedliche Art referenziert werden. Die einzigen Anhaltspunkte, die die Syntax liefert, sind:

1. Der Kontext, in dem dies geschieht (in Visual Basic `Get` – lesend, `Let` – zuweisend, `Set` – Zuweisung einer Referenz)

2. Die Tatsache, ob eine Klammerung folgt.
3. Die Tatsache, ob eine ungeklammerte Argumentenliste folgt.
4. Die Anzahl der übergebenen Argumente.

In den Fällen ohne Argumentenliste besteht die bereits erwähnte syntaktische Überschneidung von Variablen und Propertys. Bei Klammerungen kann eine Variable mit folgendem Aufruf einer Standardeigenschaft[5] oder eine Prozedur vorliegen. Bei nur einem Argument käme auch ein Arrayzugriff in Frage. Somit bleibt nur ein einziger Fall übrig, in dem tatsächlich die des Symbols bekannt ist – nämlich der Aufruf einer Prozedur mit mindestens einem Argument. Diese Unterscheidung funktioniert nur dann, wenn es sich nicht um ein einzelnes geklammertes Argument handelt.

Es gibt also keine auch nur annähernd zufriedenstelle Möglichkeit, zu erkennen, was für eine Referenzierung vorliegt. Um die Tatsache, dass ein solcher Fall vorliegt, ersichtlich zu machen und dem Betrachter zumindest die Möglichkeit zu geben, solche Fälle zu lösen, indem er die Knotentypen gegebenfalls von Hand anpasst und fehlende Kanten hinzufügt, wird ein neuer Knoten für diese Ausnahmesituation mit dem Namen `Unresolved_Symbol` eingeführt. Da die Art des Aufrufs in der Regel nicht bestimmbar ist, wird als Kantentyp derjenige genutzt, der alle möglichen Arten der Referenzierung subsumiert. Dies ist in der RFG-Kantenhierarchie `Source_Dependency`, der eine Generalisierung von sowohl `Reference` als auch `Call` ist . Eine solche Kante wird vom `Routine`-Knotentyp hin zum neuen `Unresolved_Symbol` geführt, da alle Verwendungen von einer Routine beziehungsweise ihrer Spezialisierung, der Methode, ausgehen.

Visual Basic 6	→	**RFG**	**Neu**
Verwendung eines nicht auflösbaren Symbols		`Unresolved_Symbol`-Knoten mit `Source_Dependency`-Kante	×

Das Gravis-Icon für `Unresolved_Symbol`-Knoten ist ein roter Kreis, in dem ein Fragezeichen dargestellt ist. Es wird in Abbildung 3.6 gezeigt.

Abbildung 3.6: Gravis-Icon zur Darstellung unaufgelöster Symbole

3.2.8 Visuell erstellte Programmanteile

Die mit den visuellen Werkzeugen von Visual Studio erstellen Komponenten werden, wie schon in Abschnitt 2.1.3 beschrieben, in Form des Designer-Codes im Kopf der Moduldatei gespeichert. Dieser enthält ausschließlich Deklarationen von Objekten, deren Attributen konstante Werte zugewiesen werden. Somit werden an dieser Stelle Symbole des Moduls deklariert. Ihre Aufnahme in den RFG erfolgt in Form von `Member`n des Formulars. Alle visuell definierten Komponenten sind dabei öffentlich sichtbar. Eine Besonderheit stellt hierbei die zuerst angegebene Deklaration dar, die das Formular selbst instanziiert. Visual Basic definiert auf diese

[5] Eigenschaften – also Propertys – sind wie beschrieben Prozeduren und können daher auch Parameter haben.

Weise die Standardinstanz zu jedem Formular, die im Programm als globale Variable existiert (siehe Abschnitt 2.1.3).

Diese Variable passt nicht zur sonst geltenden Semantik von Visual Basic, die in Klassen und Formularen nur instanzgebundene Member erlaubt, aber keine globalen Variablen. Es gibt in VB6 eigentlich keine statischen Variablen in Klassen. Die vorliegende RFG-Modellierung erlaubt aufgrund der Trennung des Modul- und Klassenbegriffs dennoch die Abbildung dieses Sonderfalls. Die Standardinstanz wird nicht als Teil der Klasse abgebildet, sondern als globale Variable, die Teil des *Moduls* ist. Somit wird ihre Modulzugehörigkeit korrekt abgebildet, ohne in der Modellierung ein eigentlich gar nicht vorhandenes `Static`-Konzept zu berücksichtigen.

Schwieriger ist dagegen die Abbildung der Wertzuweisungen im Designer-Code. Zwar werden an dieser Stelle keine Prozeduren aufgerufen, jedoch werden Member gesetzt, so dass eine entsprechende `Member_Set`-Kante in den RFG aufgenommen werden sollte. Allerdings gibt es keine Routine von der diese Kante ausgehen könnte, denn der Designer-Code ist nicht Teil einer Prozedur, sondern steht frei im Kopf des Formulars.

Da klar ist, dass diese Initialisierungen immer dann ausgeführt werden, wenn eine neue Instanz des Formulars erstellt wird, müssten sie am ehesten Teil eines Konstruktors sein. Solch ein Konzept gibt es, wobei die Methode stets den Namen `Form_Initialize` trägt. Die Kanten der Attributinitialisierungen im Designer-Code werden im RFG daher dieser Methode zugeschlagen. So kann der Designer-Code ohne weitere Änderungen am Modell ebenfalls abgebildet werden.

3.2.9 Das vollständige RFG-Schema für Visual Basic 6

Aus den vorhergehenden Abschnitten ergibt sich das gesamte RFG-Schema für Visual Basic 6, das bislang nur in Ausschnitten gezeigt wurde. Abbildung 3.7 zeigt es in seiner Gesamtheit.

Die Sprache kann mit nur geringfügigen Änderungen am RFG-Modell abgebildet werden. Es wurden lediglich zwei neue Knotentypen, sowie drei neue Kantentypen definiert, die das Konzept des Events sowie den Sonderfall des unaufgelösten Symbols modellieren. Weiterhin wurden sechs neue Attribute eingeführt. Die Möglichkeit dem RFG dynamisch neue Attribute hinzuzufügen, sogar ohne dass diese explizit deklariert werden müssen, macht diese Änderung jedoch unerheblich, da die Attribute allesamt rein informativen Charakter haben. Das Einfügen von Attributen hat demnach keinen wesentlichen Effekt auf das RFG-Modell. Alle weiteren verwendeten Elemente waren bereits zuvor Teil des Modells.

3.2.10 Konventionen zur Grapherstellung

Generell beinhaltet der RFG alle explizit deklarierten und globalen Programmelemente. Zusätzlich werden einige implizit vorhandene Elemente ebenfalls abgebildet. Dies ist beispielsweise für vererbte Klassenfelder der Fall. Im Fall von Visual Basic 6 enthält ein konkretes Formular demnach alle von der Klasse `Form` geerbten Member und Methoden, auch wenn diese nicht explizit dort deklariert wurden.

Ebenso enthalten sind externe Symbole aus Bibliotheken, sofern diese auch tatsächlich verwendet werden. Somit führt die bloße Einbindung einer Bibliothek in ein Visual Basic 6 Projekt nicht dazu, dass alle von der Bibliothek exportierten Symbole auch tatsächlich dargestellt werden. Bei der Erstellung des RFG sind nur diese hinzuzufügen, die aufgerufen oder in sonst irgendeiner Weise verwendet werden. Wird ein solches Element von einem anderen umgeben (im RFG durch die `Enclosing`-Kante modelliert), so ist dieses ebenfalls in den Graphen zu

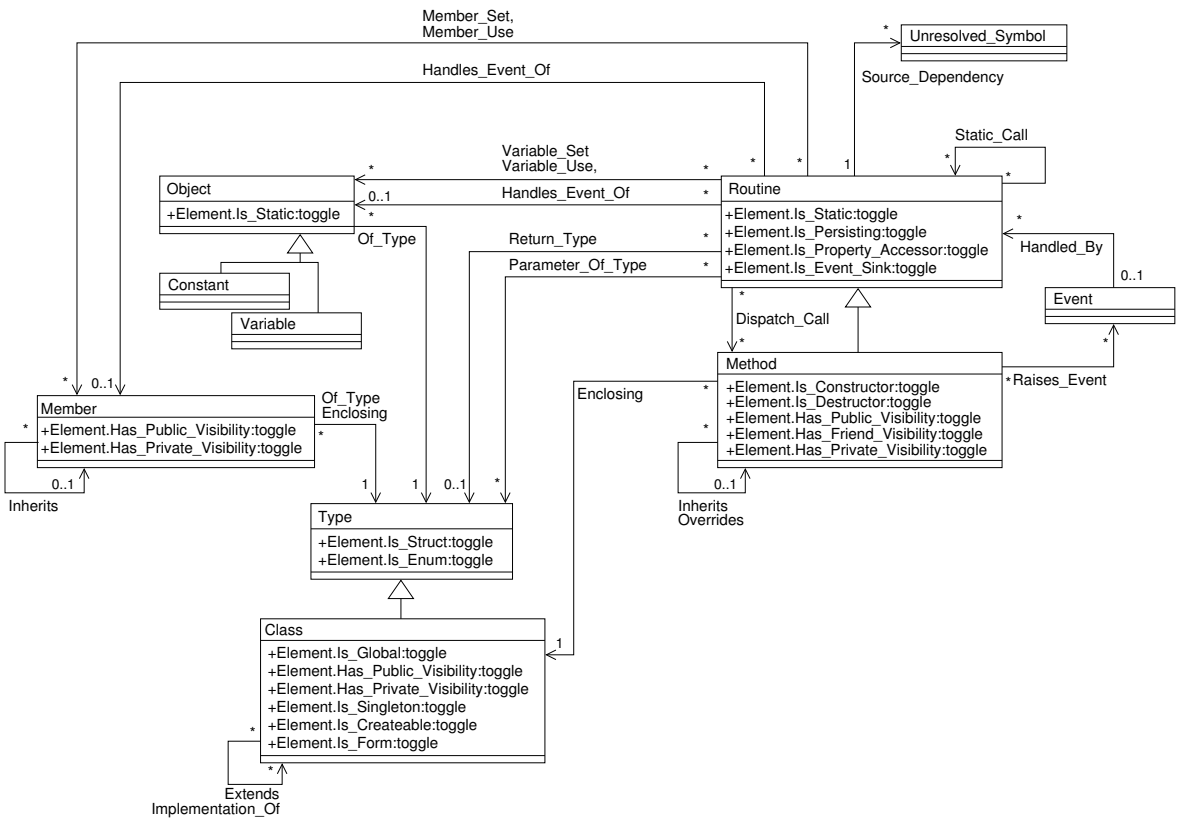

Abbildung 3.7: Das RFG-Schema für Visual Basic 6

übernehmen. Der Aufruf einer Methode hätte beispielsweise die Aufnahme von Methode und umgebender Klasse zur Folge.

3.3 Anforderungen an die Analyse von Visual Basic 6

Nachdem das RFG-Schema für Visual Basic 6 vollständig definiert ist und somit feststeht, welche Elemente der Sprache und ihrer Semantik abzubilden sind, lassen sich die konkreten Anforderungen an die Analyse der Programme aufstellen. Diese sind im Folgenden aufgelistet.

- **Analyse aller im Programm deklarierten Symbole**

  Ein Großteil der Knoten des RFG ergibt sich aus den im Programm deklarierten Symbolen. Teil des RFG sind alle Symbole, ausgenommen derjenigen, die im Kontext einer Prozedur deklariert wurden und somit lokal sind. Ihre Zugehörigkeit zu anderen Symbolen wie Klassen ist ebenfalls zu ermitteln. Ebenso sind jene Informationen zu erheben, die als Attribute and den RFG annotiert werden sollen.

- **Erkennen von Spezialfunktionen**

  Die besonderen Prozedurformen von Visual Basic sind in Form von Attributen definiert und erhalten ihre besondere Semantik durch die Verwendung bestimmter Namensmuster. Eine eventbehandelnde Prozedur wird definiert, indem ihr Name aus dem der Variable, die das eventauslösende Objekt referenziert, und dem des Events zusammengesetzt wird. Ähnlich wird bei der Implementierung von Interfaces verfah-

ren. Diese Methoden müssen anhand ihrer Namen und der gesammelten Symbolinformation des Programms erkannt werden.

- **Auflösung der Beziehungen aller Symbole**

  Die Symbolinformationen müssen weiterhin dazu genutzt werden, um die Beziehungen zwischen den einzelnen Symbolen erkennen zu können. Dies bedeutet die Ermittlung des Typs von Symbolen, um Symbole mit der `Of_Type`-Kante mit ihrem Typ zu verbinden, sofern es sich nicht um einen primitiven Typen handelt. Zudem sind die Vererbungshierarchien nachzuvollziehen, damit diese in Form von Kanten zwischen den entsprechenden Klassen, Membern und Methoden abgebildet werden können.

- **Auflösung aller Aufrufe und Variablenverwendungen**

  Die `Call`- und `Use`-Kanten des RFG stellen semantische Aspekte dar, die nur den Prozeduren zu entnehmen sind und dort auch von lokal definierten Variablen abhängen können. Daher ist es notwendig, die Symbole, die innerhalb von Prozeduren deklariert werden, in die Analyse mit einzubeziehen. Implizit deklarierte Variablen müssen dabei als solche erkannt werden. Um die Aufrufe und sonstigen Verwendungen von Symbolen erkennen zu können, muss der gesamte Quelltext auf die darin verwendeten Symbole untersucht werden. Jeder Name muss erkannt werden, um festzustellen, welches Symbol referenziert wird. Für eine korrekte Namensauflösung sind zudem Kenntnisse über den Aufbau der Namensräume und Geltungsbereiche in Visual Basic notwendig. Dies ist insbesondere für die Identifikation von implizit deklarierten Variablen von Bedeutung. Die jeweils unterschiedlichen Kantentypen erfordern es, dass der Kontext, in dem ein Symbol verwendet wird (beispielsweise lesend oder schreibend) ebenfalls festgestellt wird.

- **Analyse der von COM-Bibliotheken exportierten Symbole**

  Da auch die verwendeten Symbole externer Bibliotheken in den RFG aufgenommen werden, müssen diese ebenfalls bekannt sein.

KAPITEL 4

Analyse von Visual Basic 6

Um den Resource Flow Graph für ein Programm generieren zu können, müssen die dort abzubildenden Informationen über die Programmbestandteile zunächst gewonnen werden. In den vorangegangenen Kapiteln habe ich bereits Anforderungen erarbeitet aus denen hervorgeht, welche Daten benötigt werden. Ebenso habe ich erörtert, auf welchem Wege sie beschafft werden können. Dieses Kapitel beschreibt die Realisierung von Analysewerkzeugen, die die relevanten Daten aus den Quelltexten eines Programms und den verwendeten Bibliotheken extrahieren.

Die Untersuchung von Quelltexten und Bibliotheken dient dazu, diese Syntaktisch zu zerlegen und in ein einheitliches und zur Weiterverarbeitung geeignetes Format – beispielsweise in Form von Syntaxbäumen – zu bringen. Ist dieser lexikalische und syntaktische Teil der Analyse durchgeführt, so muss die Semantik des Programms aus den erzeugten Datenstrukturen ausgelesen werden, um die für den RFG benötigten Informationen zusammenzustellen.

Somit teilt sich die Analyse von Visual Basic 6 in die drei wesentlichen Teilbereiche der Quelltext-, COM- und der semantischen Analyse auf und folgt damit im Wesentlichen dem Aufbau eines *Compiler-Frontends* wie es in [ASU86] beschrieben wird. Abbildung 4.1 zeigt die Zusammenhänge dieser Aufgabenbereiche und der verwendeten und erhobenen Daten in grober Darstellung.

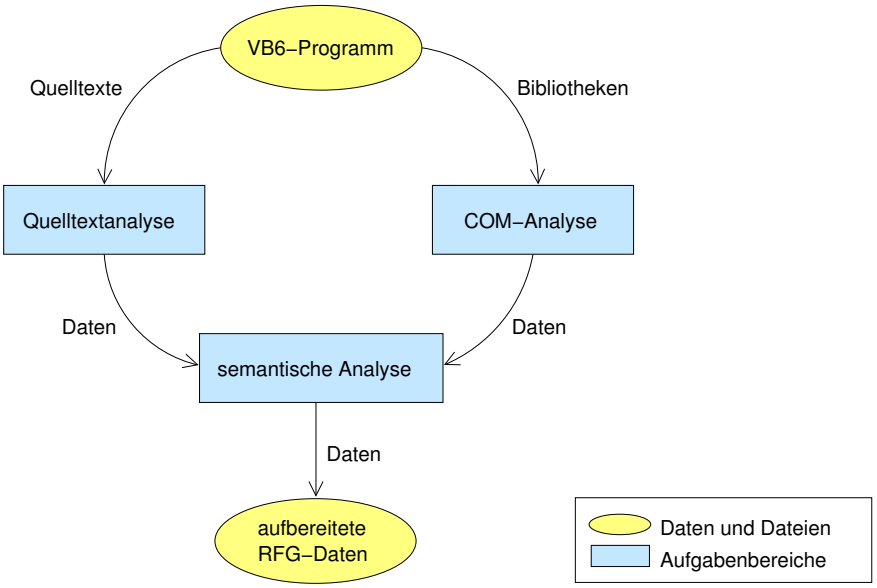

Abbildung 4.1: Daten und Aufgaben der VB6-Analyse

Die Generierung des RFG ist in Abbildung 4.1 nicht berücksichtigt. Sie stellt zwar die Anfor-

derungen für die Analyse, ist aber nicht als Teil davon zu verstehen. Vielmehr ist es für eine eventuelle spätere Weiterentwicklung der VB6-Analyse, beispielsweise zur IML-Generierung, sinnvoll, sie nicht mit Aufgaben der RFG-Generierung zu durchmengen. Daher liegt das Ziel der VB6-Analyse darin, die benötigten Daten aus den Programmen zu extrahieren und in einem möglichst allgemeinen Datenformat zur Verfügung zu stellen. Für diesen Zweck bieten abstrakte Syntaxbäume sowie eine Symboltabelle (vgl. [ASU86]) eine sinnvolle Grundlage. Die RFG-Generierung, welche die Ergebnisdaten der VB6-Analyse nutzt, wird in Kapitel 5 behandelt.

Die Aufteilung dieses Kapitels orientiert sich an den soeben dargestellten Aufgabenbereichen. Zunächst werde ich die Quelltextanalyse näher betrachten. Ein grundlegendes Problem hierbei sind die unvollständigen Grammatikdefinitionen, die für die notwendigen Untersuchungen nicht ausreichen. Daher werde ich zunächst Möglichkeiten vorstellen, wie mit einer solchen Ausgangssituation umgegangen werden kann und dann das von mir gewählte Vorgehen daraus ableiten und seine Umsetzung beschreiben.

Die Analyse von COM-Bibliotheken ist dagegen ein vorwiegend technisches Problem, dessen Lösung die Spezifikation eines Austauschformates für Schnittstellendefinitionen beinhaltet. Abschließend beschreibe ich die semantische Analyse, die ebenfalls durch das unvollständige Wissen über die Semantik der Sprache verkompliziert wird. Die zentrale Aufgabe dieses Bereichs ist das Erkennen von Symbolen und die Auflösung ihrer Beziehungen im Programm. Die dazu nötigen Grundlagen, die zum Teil nicht ausreichend dokumentiert sind, werde ich dort herleiten und anwenden, um die Ausgangsdaten für die RFG-Generierung zu erzeugen.

In diesem Kapitel mache ich intensiv von Begriffen aus der Syntaxanalyse Gebrauch. Zu ihrer Erklärung sei vorab auf [ASU86] verwiesen.

Kapitelinhalt

4.1 Quelltextanalyse

Der übliche Weg Quelltexte zu analysieren besteht nach [ASU86] darin, einen Parser auf Grundlage einer formalen Grammatik zu erstellen oder anhand eines Parsergenerators zu erzeugen. Folglich ist dies nur möglich, wenn die Grammatik auch bekannt ist – eine Voraussetzung, die für diese Arbeit nicht gegeben ist. Daher muss zunächst ein Weg gefunden

werden, wie trotz dieser Ausgangssituation eine möglichst akkurate Analyse durchgeführt werden kann, die den zuvor erarbeiteten Anforderungen gerecht wird.

Ich werde in diesem Abschnitt Möglichkeiten dazu vorstellen und ein eigenes Vorgehen, das an die Rahmenbedingungen dieser Arbeit angepasst ist, wählen und beschreiben. In diesem Zuge behandle ich auch die verwendeten Hilfswerkzeuge. Das Ergebnis – eine neue Visual Basic 6 Grammatik – werde ich jedoch nicht im Detail beschreiben, da dies den Rahmen dieses Dokuments sprengen würde. Stattdessen beschränke ich mich auf wenige Beispiele, die typische Probleme beschreiben, die in der Realisierung berücksichtigt werden mussten. Eine auf Lesbarkeit optimierte Fassung der Grammatik kann dem Anhang entnommen werden. Zudem ist die vollständige Grammatikdefinition Teil der digitalen Abgabe.

4.1.1 Sprachanalyse mit unvollständigem Wissen

In der Vergangenheit wurden verschiedene Ansätze vorgeschlagen, um im Reverse Engineering mit Sprachen umzugehen, die wegen einer unvollständigen Dokumentation oder aus ähnlichen Gründen schwer zu analysieren sind. Diese Schwierigkeiten können wie in diesem Fall aus dem Fehlen einer vollständigen und korrekten Grammatikdefinition erwachsen aber auch ganz andere Ursachen haben, wie beispielsweise vielfältige Dialekte einer Sprache, in Webseiten eingebettete Quelltexte oder Fehler in den Programmen. In diesem Abschnitt stelle ich zwei unterschiedliche Herangehensweisen vor.

4.1.1.1 Partielle Analysemethoden

Eine Möglichkeit Quelltexte trotz unvollständigen Wissens über ihre Syntax zu analysieren, besteht darin, gezielt nur die Teile zu betrachten, deren Syntax bekannt ist, die für die anzustellende Analyse von Interesse sind und alles weitere zu ignorieren. Zur Generierung eines Aufrufgraphen zu einem Programm, wäre es beispielsweise ausreichend, die Funktionsdeklarationen und die gegenseitigen Aufrufe innerhalb der Funktionen zu betrachten und alles weitere vollkommen zu ignorieren. Dinge wie Kontrollstrukturen oder Import-Anweisungen könnten hier vernachlässigt werden.[1] Viele andere Einsatzszenarien sind denkbar.

Solche Verfahren wurden von Gail Murphy und David Notkin als *Lightweight Source Model Extraction* (in [MN95]) oder von Leon Moonen in Form von *Island Grammars* (in [Moo02]) vorgeschlagen. Moonen definiert seine Grammatiken folgendermaßen:

> *„An island grammar is a grammar that consists of two parts: (i) detailed productions that describe the language constructs that we are particularly interested in (so called **islands**), and (ii) liberal productions that catch the remainder of the input (so called **water**).“*

Die Verfahren sind sich dabei sehr ähnlich. Durch die Verwendung von kontextfreien Grammatiken zur Spezifikation der Inseln erreichen die Island Grammars jedoch eine größere Genauigkeit als das rein lexikalische Verfahren von Murphy und Notkin.

Interessant an diesen Verfahren ist die Möglichkeit, Quelltextanalysen auch mit sehr begrenztem Wissen über die Sprache durchzuführen. Es wird kein Wissen über die Teile der Syntax benötigt, die keine für die Analyse relevanten Informationen enthalten. Zudem ist die Lösung sehr robust, da sie sogar mit fehlerhaften Quelltexten problemlos zurecht kommt. Entspricht

[1] Zumindest wenn man sich damit begnügt statische Aufrufe zu betrachten.

etwas nicht den gesuchten Mustern der Inseln, so wird es schlicht ignoriert. In Bezug auf Visual Basic 6 wäre dies von Vorteil, da Sonderformen und undokumentierte Teile der Sprache nicht behandelt werden müssten.

Diese Ansätze sind jedoch nicht exakt, sondern approximativ. Es ist möglich, dass vorhandene und gesuchte Sprachelemente nicht erkannt werden (falsche Negative) oder eigentlich irrelevanter Quelltext als relevante Information missinterpretiert wird (falsche Positive). Zudem ist die Bewertung der Analyseergebnisse schwierig, da die Angabe falscher oder ungenauer Muster zu keinen nachvollziehbaren Fehlern im Analysevorgang führt. Der Abgleich von Ergebnissen und Vorgabe muss daher rein manuell erfolgen. Gerade dann, wenn das Wissen über die Syntax lückenhaft ist, besteht folglich ein großes Risiko die Inseln ungenügend genau zu spezifizieren, wobei solche Fehler zugleich schwer auszumachen sind.

Solche partiellen Analysemethoden erscheinen zudem nur dann sinnvoll, wenn auch tatsächlich ein größerer Teil der Quelltexte ignoriert werden kann. Gemäß den erarbeiteten Anforderungen für die Analyse von Visual Basic sind unter anderem folgende Aspekte der Syntax von Bedeutung:

- Alle Deklarationen auf Modulebene (Prozeduren, Variablen, etc.)
- Variablen- und Memberverwendungen inklusive ihres Typs (lesend oder schreibend)
- Alle Verzweigungen im Kontrollfluss (zur McCabe-Berechnung)
- Alle Bezeichner und sonstigen Wörter und Zeichen (zur Halstead-Berechnung)
- Modulattribute (für Klassen-Attribute wie `Is_Creatable`)

Schon diese allgemeine Betrachtung verdeutlicht, dass die Anforderungen an die Analyse kaum zulassen, große Teile der Quelltexte zu ignorieren. Neben Deklarationen, Bezeichnern (Symbolverwendungen), ihrem Kontext (Anweisungen) und dem Kontrollfluss bleibt wenig übrig, das sich ignorieren ließe. Zwar gibt es in der Syntax durchaus auch Teile, die für die RFG-Generierung unbedeutend sind – beispielsweise konstante Werte in Form von Zahlen, Zeichenketten oder ähnlichem – allerdings sind gerade diese eher unkomplizierte Aspekte der Syntax, deren Beschreibung kein wirkliches Problem darstellt. Die Schwierigkeiten, die schon im Vorfeld absehbar sind – beispielsweise die unklare Trennung von Bezeichnern und Schlüsselwörtern oder die diversen Kurzformen zum Prozeduraufruf – beziehen sich auf Teile der Syntax, die zwingend analysiert werden müssen, um die benötigten Daten zur extrahieren.

Hinzu kommt, dass es für die Berechnung der Halstead-Metrik unabdingbar ist, Bezeichner und Schlüsselwörter unterscheiden zu können (vgl. Abschnitt 2.4.4). Die Trennung von Bezeichnern und Schlüsselwörtern ist in Visual Basic wie in Abschnitt 2.1.3 beschrieben nicht strikt umgesetzt, sondern unterliegt je nach Kontext unterschiedlichen Regeln. Hierdurch ergeben sich viele Situationen, in denen die Art eines Tokens erst durch Berücksichtigung des Kontextes und der dort geltenden Regeln erkannt werden kann. So ist es nicht möglich, eine Insel zu definieren, die sich durch die Zusammensetzung aus Bezeichnern kennzeichnet, da auf Basis der Lexeme allein nicht in jedem Fall eindeutig entscheidbar ist, ob ein Lexem ein Schlüsselwort oder einen Bezeichner darstellt.

Offenbar sind die Anforderungen, die die Generierung des RFG und die Erhebung von Metriken stellen, zu umfangreich, um mit einem partiellen Ansatz wie den Island Grammars sinnvoll untersucht zu werden. Zumindest ist der Vorteil, der sich durch das Ignorieren bestimmter Teile der Programme ergeben soll, in diesem Fall als eher gering einzuschätzen. Der überwiegende Teil der Syntax muss von der Grammatik erkannt werden und der Einsatz der

hier beschriebenen Techniken würde daher vermutlich einen übermäßigen Aufwand für einen eher kleinen Effekt bedeuten.

Schließlich ist es zudem wünschenswert, dass die Quelltextanalyse zu einem späteren Zeitpunkt für eine IML-Generierung erweitert werden kann. Diese erfordert eine noch detailliertere Analyse der Quelltexte, so dass Island Grammars hier keine sinnvolle Option bieten. Insgesamt sind sie in Anbetracht der Anforderungen dieser Arbeit kein angemessenes Mittel.

4.1.1.2 Rekonstruktion von Grammatiken

Ralf Lämmel und Chris Verhoef beschreiben in [LV01] einen eher konventionellen Ansatz auf Grundlage etablierter Techniken der Syntaxanalyse und des Compilerbaus, wie sie in [ASU86] beschrieben sind. Sie schlagen ein allgemeines Vorgehen zur Erstellung von Grammatiken für existierende Sprachen, zu denen diese formale Beschreibung nicht verfügbar oder unvollständig ist, vor. Als Grundlage dienen dabei Handbücher oder sonstige technische Beschreibungen sowie Quelltexte der zu untersuchenden Sprache.

Davon ausgehend, dass diese notwendigen Voraussetzungen vorhanden sind, eine formale Grammatik dagegen aber nicht, teilen sie die Rekonstruktion in die folgenden Phasen ein:

1. **Extraktion einer Roh-Grammatik:** Syntaxdefinitionen werden in einem automatisierten Prozess aus Handbüchern und sonstiger Dokumentation ausgelesen.

2. **Verbindung der Grammatikregeln:** Die Referenzierungen zwischen den einzelnen und in der Dokumentation getrennt beschriebenen Produktionen werden gesucht und aufgelöst, indem Namen vereinheitlicht werden und Verknüpfungen zwischen den einzelnen Produktionen hergestellt werden.

3. **Testbasierte Korrektur:** Die rekonstruierte Grammatik wird getestet, indem ein Parser zur Fehlerkorrektur erstellt wird. Mit ihm wird versucht, Beispielprogramme der Sprache zu erkennen. Auftretende Fehler werden nach dem Versuch-und-Irrtum-Prinzip schrittweise korrigiert bis keine weiteren mehr auftreten.

Die Ausgangssituation meiner Arbeit lässt sich hier sehr gut einordnen. Es sind bereits (partielle) Grammatikdefinitionen vorhanden, allerdings sind diese unvollständig und müssen mit Wissen erweitert werden, dass nicht dokumentiert ist. Entstanden sind die Grammatiken nach Prinzipien, die sich mit den ersten beiden Schritten aus dem hier vorgestellten Verfahren decken. Als Grundlage dienten Informationen aus den Handbüchern und der Online-Referenz entnommen, wenn auch in einem vermutlich manuellen Prozess. Die Schritte eins und zwei sind in diesem Fall also bereits vorgeleistet. An ihre Stelle würde die Transformation der bestehenden Grammatiken in die Beschreibungssprache des für diese Arbeit gewählten Parsergenerators (ein Generator, der C++-Code erzeugen kann) treten. Zudem müssen die Teile der Sprache, die von den bestehenden Grammatiken nicht abgedeckt werden, hinzugefügt werden. Dies kann nur nach dem Versuch-und-Irrtum-Prinzip erfolgen, da es keinen anderen Zugang zu dem fehlenden Wissen über die Syntax gibt.

Das Testverfahren von Lämmel und Verhoef basiert auf einem speziellen Parser, der eigens zur Suche von Fehlern in der Grammatikdefinition erstellt wurde. Wesentlich ist hierbei, dass ein Top-Down-Algorithmus gewählt wurde, um die Nachvollziehbarkeit von Fehlern zu verbessern. Bottom-Up-Verfahren wie beispielsweise der LR-Algorithmus (siehe [ASU86, S. 215ff]) bedienen sich abstrakter Verfahren, die auf Tabellen, Stacks und Operationen über selbige basieren. Tritt dabei ein Fehler auf, so ist er weniger intuitiv nachvollziehbar als es in

einem *Recursive Descent Parser*, der zu jeder Grammatik-Produktion eine Funktion enthält, der Fall wäre (vgl. [ASU86, PQ95]).

Im Gegensatz zu den beiden Methoden, die ich zuvor vorgestellt habe, zielt dieses Vorgehen auf die Generierung einer vollständigen Grammatik ab. Eine vollständige und korrekte Grammatik, in dem Sinne, dass sie die gleiche Sprache erzeugt wie die tatsächliche VB6-Grammatik, kann aber auch dieser Ansatz nicht mit Sicherheit liefern. Das testbasierte Vorgehen erlaubt nur eine Annäherung an die tatsächliche Grammatik, indem es begrenzten Aufschluss darüber gibt, was korrekte Programme der Sprache sind, nicht aber darüber, welche Programme *nicht* korrekt sind.

Rein formal betrachtet ist die von einer kontextfreien Grammatik[2] G erzeugte Sprache L nach [HU79]:

$$L(G) = \{w | w \in T^*, S \overset{*}{\Rightarrow} w\}$$

Hierbei ist w ein Wort der Sprache und T^* die Menge aller Wörter über die terminalen Symbole. Demnach ist die Sprache einer Grammatik also die Menge aller, unter Verwendung korrekter Ableitungen, anhand ihrer Produktionen ableitbaren Wörter, die nur aus terminalen Symbolen bestehen. Im Sinne einer Programmiersprache sind die Wörter der Sprache einzelne Programme. Will man eine Grammatik definieren, die ebenfalls L erzeugt, also eine *äquivalente* Grammatik, so müsste diese alle Programme der Sprache erzeugen können und keine weiteren. Um nachweislich Vollständigkeit und Korrektheit zu erreichen, müssten in dem hier vorgestellten Verfahren konsequenterweise auch alle möglichen Programme untersucht werden. Dass dies nicht möglich ist, liegt auf der Hand.

Stattdessen wird lediglich eine Grammatik erstellt, die die Menge der herangezogenen Testprogramme nachweislich erzeugen (oder durch einen Parser erkennen) kann. Dass die von einem Parser erzeugten Ableitungen und damit auch die Ableitungsbäume korrekt sind, lässt sich ebensowenig sicherstellen. Hier kann lediglich davon ausgegangen werden, dass die Handbücher und das Wissen über Programmiersprachen dabei hilft, Produktionen zu wählen, die korrekte Ableitungen herbeiführen, also die Lexeme der Sprache syntaktisch korrekt strukturieren. Anders als die Frage, ob die Grammatik ein bestimmtes Programm erzeugen kann, lässt sich die Korrektheit des erzeugten Syntaxbaumes nicht direkt nachweisen.

Es ist also nicht zu erwarten, dass durch das testbasierte Verfahren eine Grammatik hervorgebracht wird, die zu der tatsächlichen VB6-Grammatik äquivalent ist. Allerdings ist dies auch nicht zwingend notwendig. Die Anforderungen an eine Grammatik sind stark von ihrem Verwendungszweck abhängig. Lämmel und Verhoef stellen dazu eine Klassifizierung für Grammatiken auf, die Programme der gleichen Sprache für unterschiedliche Zwecke modellieren. Während es zu den Aufgaben eines Compilers gehört, fehlerhafte Quelltexte abzuweisen, kann im Reverse-Engineering von Programmquelltexten in der Regel davon ausgegangen werden, dass ausschließlich korrekte Programme, die vom Compiler akzeptiert werden, Gegenstand der Untersuchung sind. Von daher ist es gar nicht nötig entscheiden zu können, ob eine Eingabe ein korrektes Programm ist oder nicht. Der Parser unterliegt weniger strikten Anforderungen und darf auch syntaktisch falsche Programme akzeptieren.

Dagegen ist es wünschenswert, dass alle *korrekten* Programme durch den Parser erkannt und korrekt in einen Baum transformiert werden können. Dies lässt sich wie bereits erläutert nicht garantieren. Es ist nur möglich sich diesem Ziel durch eine möglichst große und heteroge-

[2] Eine Grammatik $G = (V, T, P, S)$ besteht nach [HU79, S. 83] aus einer Menge von Variablen V, einer Menge von terminalen Symbolen T - dies sind die einzelnen Wörter und Zeichen der Sprache - einer Menge von Produktionen P sowie einer Strartproduktion S.

ne Menge an Testfällen anzunähern. Lämmel und Verhoef beschreiben, dass die Anzahl der Fehler, die in erstmalig untersuchten Programmen auftreten, bei ihrem Vorgehen mit der voranschreitenden Verfeinerung der Grammatik stark abnimmt. Dies zeigt, dass eine Annäherung an die tatsächliche Grammatik der Sprache erfolgt.

4.1.2 Vorgehensweise in dieser Arbeit

In [LV01] wird letztendlich nur ein Vorgehen unter Verwendung bekannter Techniken beschrieben und seine praktische Anwendbarkeit durch eine Fallstudie am Beispiel eines COBOL-Dialekts gezeigt. Die Methodik lässt sich auch für die Aufgaben dieser Arbeit einsetzen, wobei die ersten Schritte bereits vorgeleistet wurden. Es fehlt noch eine weitere testbasierte Verfeinerung der Grammatikdefinition wie sie Lämmel und Verhoef in Schritt drei ihres Vorgehens durchführen.

Die Erstellung der VB6-Grammatik erfordert in dieser Arbeit also die folgenden Schritte:

1. Wahl eines Parsergenerators gemäß den Anforderungen.
2. Transformation des vorhandenen Wissens und der Grammatiken in das Grammatikformat des gewählten Generators.
3. Testbasierte Verfeinerung und Korrektur der Grammatik.

Ich werde diese Schritte im Folgenden einzeln beschreiben.

4.1.2.1 Parsergenerator

Parser für komplexe Sprachen werden üblicherweise nicht von Hand sondern unter Zuhilfenahme eines Generators, der Parserprogramme aufgrund formaler Grammatikdefinitionen automatisch erstellt, implementiert [ASU86]. In dieser Arbeit kommt *ANTLR*, das in [PQ95] beschrieben wird, in der Version 2.7.2 zum Einsatz. Der Name steht für „**A**nother **T**ool for **L**anguage **R**ecognition“. Dieser Generator erzeugt Lexer und Parser, die einen $LL(k)$-Algorithmus mit beliebigem k verwenden. Zudem erlaubt er den Einsatz von semantischen und syntaktischen Prädikaten. Mit diesen kann die Anwendbarkeit von Alternativen in den Produktionen der Grammatik gesteuert und vom Status des Parsers oder einem expliziten Lookahead beliebiger Länge abhängig gemacht werden. Diese Erweiterung des klassischen $LL(k)$ wird in [PQ95] als *pred-LL(k)* bezeichnet. Neben Java kann unter anderem auch C++-Code erzeugt werden.

Es gibt unterschiedliche Parsergeneratoren, die sich zum Teil anderer Algorithmen bedienen, wie beispielsweise *YACC*[3], das *LALR*-Parser[4] erzeugt oder *JavaCC*, das sich ebenfalls $LL(k)$ bedient. Die Wahl fiel insbesondere deswegen auf ANTLR, weil es die oben bereits erwähnten *Recursive Descent Parser* in der vorgegebenen Programmiertsprache erzeugt. Somit erleichtert es die testbasierte Korrektur der Grammatik. Fehler können recht unkompliziert und intuitiv anhand des Aufrufstacks im Parser zurückverfolgt werden.

ANTLRs Prädikate bieten zudem eine große Flexibilität, indem sie bei Bedarf beispielsweise einen expliziten Lookahead von beliebiger Länge, unabhängig vom definierten k, erlauben. Die semantischen Prädikate erlauben es darüber hinaus den Parse-Prozess durch semantische Informationen zu steuern. Vor dem Hintergrund der teilweise uneinheitlichen und wenig strikten

[3] YACC: http://dinosaur.compilertools.net/
[4] Für die Beschreibung verschiedener Parsing-Algorithmen siehe [ASU86]

VB6-Syntax ist eine möglichst große Flexibilität seitens des Parsergenerators wünschenswert. Da Teile der Syntax und der Semantik zudem noch gar nicht bekannt sind, ist im Vorfeld nicht abzusehen, welche Schwierigkeiten bei der Analyse noch auftreten können. Auch daher sollte ein Parser möglichst flexibel sein, um bessere Aussichten für die Lösung bislang gar nicht bekannter Probleme zu bieten und zu vermeiden, dass die Grammatikdefinition in einer Sackgasse endet, weil ein unerwartetes Problem mit dem gewählten Parsergenerator nicht oder nur schwer zu lösen ist.

Ein weiterer Vorteil besteht zudem darin, dass der Testparser, anders als in Lämmels und Verhoefs Studie, direkt weiterverwendet und zur Quellcodeanalyse eingesetzt werden kann. ANTLR ermöglicht es, automatisch abstrakte Syntaxbäume zu generieren, deren Struktur und Inhalt durch eine spezielle Syntax direkt in den Grammatikproduktionen manipuliert werden kann.

Ein Nachteil von $LL(k)$ und damit auch von ANTLR besteht darin, dass es die Eliminierung von Linksrekursion in der Grammatik erfordert (siehe [ASU86]). Dieser Nachteil relativiert sich jedoch durch die Tatsache, dass die vorliegenden Grammatiken für JavaCC und TXL ebenfalls $LL(k)$-Grammatiken sind, also bereits Lösungen für einen Großteil der auftretenden Linksrekursion bieten.

Ein wenig unorthodox erscheint es, dass ANTLR seine Lexer ebenfalls mit dem $LL(k)$-Algorithmus realisiert, anstatt reguläre Ausdrücke in endliche Automaten zu transformieren wie es sonst übliche Praxis ist. Im $LL(k)$-Lexing werden die einzelnen Zeichen des Zeichenstroms betrachtet. Anhand der im fixen Lookahead befindlichen Zeichen wird entschieden, welche Produktion gewählt wird.

Die wesentlichen Anforderungen an den Parsergenerator sind aufgrund des gewählten Vorgehens jedoch die Flexibilität und die möglichst einfache Nachvollziehbarkeit von Fehlern. Zusammen mit der Tatsache, dass ANTLR C++-Quelltexte generieren kann, bietet es somit gute Voraussetzungen für die angestrebte Implementierung.

4.1.2.2 Transformation der Grammatiken zu ANTLR

Die Sprachen, die JavaCC und ANTLR zur Grammatikdefinition einsetzen, sind ähnlich, da beide Generatoren erweiterte BNF-Definitionen verwenden. Es gibt jedoch eine Vielzahl von Details, angefangen von der Syntax einer Produktion bis hin zu den Prädikaten (die auch JavaCC in ähnlicher Form kennt) die auf unterschiedliche Weise ausgedrückt werden. Beide Definitionsformen sind sehr implementierungsnah. Die Lesbarkeit und damit auch Verständlichkeit der JavaCC-Grammatik leidet dabei unter der stark mit Java-Code durchsetzten Definition. Zudem hat der Autor Paul Cager viele Probleme nicht in der Grammatik, sondern durch Java-Code gelöst. Dabei werden Einschränkungen des Algorithmus umgangen. Die Verständlichkeit leidet allerdings stark unter diesen Workarounds.

In der praktischen Umsetzung einer ANTLR-Fassung, die von Hand erfolgte, hat sich herausgestellt, dass die Syntaxanteile, die in der Referenz beschrieben werden, einfacher dort entnommen werden können als aus der teils undurchsichtigen JavaCC-Grammatik. Somit diente die zur Verfügung stehende Dokumentation ([Mic06b] sowie die Bücher [Mon01, Mas99, Kof00, Cow00, Hau99]) als Grundlage für einzelne Kontrollstrukturen und Deklarationen, während die JavaCC-Grammatik weitergehendes Wissen über deren Verbindungen und die Form von Anweisungen sowie undokumentierter Sprachteile lieferte. Der Lexer musste wegen des vollkommen anderen Algorithmus, den ANTLR verwendet, von Grund auf neu geschrieben werden.

Für den überwiegenden Teil der Syntax war dieses Vorgehen recht problemlos. Wie in Ab-

schnitt 2.1.4.1 beschrieben, umgehen die vorliegenden Grammatiken jedoch einige umständlich abzuleitenden Teile der Syntax. Zum Beispiel, indem eigentlich komplexe Satzformen als ein Knoten im Syntaxbaum betrachtet werden. Diese Hilfskonstruktionen wurden nicht in die neue Grammatik übernommen, sondern bewusst ausgelassen, um beim späteren Testen Fehler an diesen Stellen hervorzurufen und dann eine Verfeinerung der Grammatik an den entsprechenden Stellen durchführen zu können.

4.1.2.3 Testbasierte Verfeinerung

Für die testbasierte Korrektur der Grammatik wurde die standardmäßige Fehlerbehandlung von ANTLR-generierten Parsern – die nach einem Fehler schlicht so lange versucht eine auf den aktuellen Zustand passenden Tokensequenz zu finden, bis eine solche entdeckt wird oder der Tokenstrom endet – durch die Ausgabe einer einfachen Fehlermeldung ersetzt. Jeder Fehler führt dabei zum Abbruch des Analysevorgangs der aktuellen Datei, da einmalige Fehler während des Parsens zu Folgefehlern führen können, die falsche Positive darstellen.

Für einen effizienten Ablauf der testbasierten Verfeinerung ist es wichtig aussagekräftige Fehlermeldungen auszugeben, die im Idealfall bereits ausreichen, um das Problem zu erkennen. Hierzu wurde das bei vielen Compilern übliche Verfahren gewählt neben der Quelltextposition auch die Programmzeile, in der das Problem auftrat, auszugeben. Darüber hinaus werden die Typen der nächsten drei Token im Lookahead ausgegeben, um ein Bild von der Repräsentation zu erhalten, die der Lexer geliefert hat. Auf diese Weise ließen sich Detailfehler, wie beispielsweise Typkürzel an Stellen, an denen sie zunächst nicht erwartet wurden, in der Regel sehr einfach identifizieren und beheben.

Lämmel und Verhoef sind ihrer Beschreibung zufolge so vorgegangen, dass sie zunächst versucht haben, ein Programm fehlerfrei zu parsieren, um danach selbiges mit weiteren Programmen durchzuführen. Ich habe dagegen den gesamten Beispielquelltext in einem Testdurchgang untersucht und alle auftretenden Fehler – pro Modul kann es davon nur einen geben – protokolliert. Als Ergebnis liefert somit jeder Test eine Liste von „echten“ Fehlern, die frei von Folgefehlern ist. Dies hat sich als hilfreich erwiesen, um Probleme, die durch die Betrachtung nur einer einzelnen Meldung nicht verständlich wurden, nachzuvollziehen. Da stets über 800 Dateien untersucht wurden, konnte bei solchen unklaren Problemen nach ähnlichen Fehlern gesucht werden. Oftmals hat dabei die Betrachtung mehrerer Ausprägungen des gleichen Fehlers geholfen, das eigentliche Problem zu verstehen.

Ein bedeutender Vorteil dieser Testweise besteht zudem darin, dass Änderungen an der Grammatik, die wiederum zu neuen Fehlern in zuvor bereits erfolgreich geparsten Dateien führen können, immer gegen den gesamten Quelltextbestand getestet werden. Dem nachträglichen Einschleichen von neuen Fehlern in die Grammatik kann so vorgebeugt werden. Mit abnehmender Fehlerzahl wird diese Vorgehensweise allerdings zunehmend zu einem zeitaufwändigen Prozess, da die Codegenerierung und die folgende Kompilation des Parsers nach jeder Änderung aufgrund der immer größer werdenden Grammatik länger dauert. Selbiges gilt für die Ausführung der Tests. Sie führen immer später zu Fehlern, wodurch ein immer größerer Teil der Quelltexte geparst wird bevor aufgrund eines Fehlers abgebrochen wird.

4.1.2.4 Detaillierte Übersicht der VB6-Analyse

Die Festlegung des Analyseverfahren erlaubt nun eine technische Sicht der VB6-Untersuchung. Am Anfang dieses Kapitels habe ich bereits die einzelnen Aufgabenbereiche und die Eingangs- sowie Ausgangsdaten grob festgelegt. Abbildung 4.2 konkretisiert diese allgemeine Orientie-

rung, indem sie die notwendigen Systemkomponenten für die VB6-Analyse festlegt und die Art ihrer Implementierung benennt.

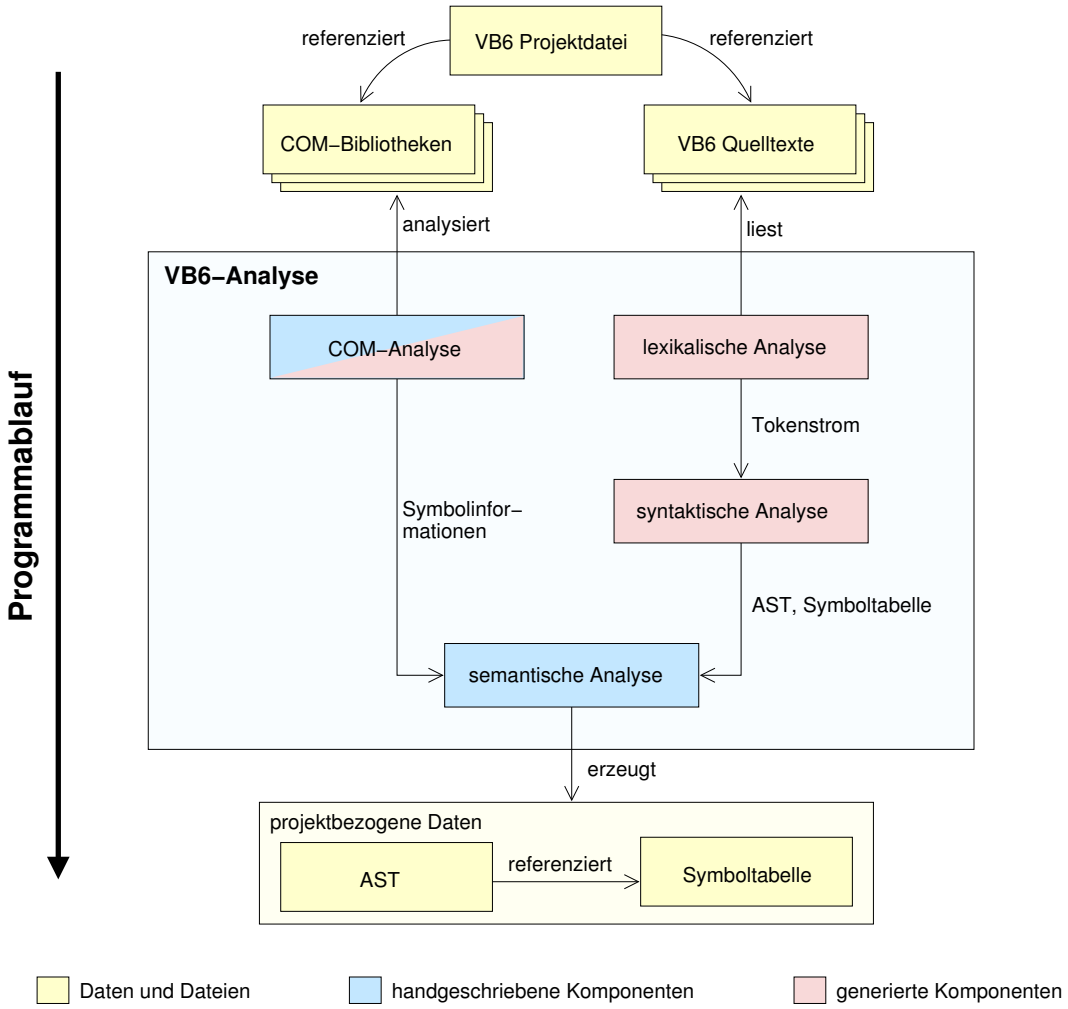

Abbildung 4.2: Technische Übersicht der VB6-Analyse

Generell basiert die Analyse auf einem VB6-Projekt, das eine übersetzbare Einheit aus einem oder mehreren Modulen bildet. Hierbei ist die Projektdatei der zentrale Ort, an dem alle relevanten Informationen über das Projekt zusammengeführt werden. Sie enthält Verweise auf die verwendeten Bibliotheken und Modulquelltexte. Diese für die Analyse relevanten Daten werden von einem einfachen handgeschriebenen Parser ausgelesen und systemintern in einer Datenstruktur vorgehalten, die es den einzelnen Komponenten erlaubt, benötigte Informationen über das Projekt abzufragen (dieser triviale Systemteil ist nicht im Bild dargestellt und wird auch nicht weiter im Detail beschrieben).

Zur COM-Analyse ist anzumerken, dass diese zweistufig verläuft. Zunächst werden die Schnittstellen der COM-Komponenten auf einem Windows-System mit einem handgeschriebenen Tool ausgelesen und in ein Austauschformat transformiert. Dieses wird dann auf dem Linux-System, auf dem die Analyse durchgeführt wird, wieder geparst und in Symbolinformationen umgewandelt. Dieser Teil erfolgt durch einen ANTLR-generierten Parser.

Am Ende der Analyse steht ein abstrakter Syntaxbaum, der die Syntaxbäume der einzelnen Quelltexte unter einer gemeinsamen Wurzel vereint sowie eine Symboltabelle mit allen im Projekt deklarierten und von den Bibliotheken exportierten Symbolen. Alle Bezeichner im

AST sind dabei ihren Symbolen zugeordnet, indem sie auf die entsprechenden Einträge der Symboltabelle verweisen. Diese Datenstruktur enthält sämtliche für den RFG benötigten Daten, wodurch die spätere RFG-Generierung lediglich eine Transformation dieser Daten in ein anderes Format darstellen wird.

Im Folgenden werde ich jedoch zunächst die erstellten Grammatiken für Lexer und Parser beschreiben und damit auf die beiden entsprechenden Komponenten der VB6-Analyse eingehen. Da es aus Platzgründen nicht möglich ist Lexer, Parser und ihre Grammatiken vollständig zu beschreiben, werde ich im Folgenden selektiv die wichtigsten Aspekte daraus vorstellen. Diese Beispiele dienen als Grundlage für später auftretende Probleme in der semantischen Analyse. Zudem sollen sie dabei helfen, die mitunter eigenwillige Syntax und Semantik der Sprache und damit die Schwierigkeit der Analyse zu illustrieren. Wie ich bereits erwähnt habe, können die Grammatikdefinitionen in ihrer Gänze dem Anhang und der digitalen Abgabe entnommen werden.

4.1.3 Lexikalische Analyse

Die lexikalische Analyse dient dazu, Programmquelltexte in ihre einzelnen Bestandteile wie Schlüsselwörter, Bezeichner und strukturgebende Zeichen zu zerlegen und diese zu klassifizieren. Diese einzelnen und als Token bezeichneten Teile dienen dem Parser als Eingangsdaten und ermöglichen es die einzelnen Lexeme, aus denen ein Programm besteht, dort abstrakt anhand der Klassifizierung zu betrachten und ihre syntaktische Zusammenstellung zu untersuchen.

ANTLR verwendet wie zuvor erwähnt den $LL(k)$-Algorithmus zu diesem Zweck. Das erfordert zwar das Entfernen von Linksrekursion in der Lexer-Grammatik, ermöglicht auf der anderen Seite aber bereits an dieser Stelle die Vorzüge eines Parsers, wie beispielsweise Actions oder sogar die syntaktischen und semantischen Prädikate, zu verwenden. Da die Lexer der beiden vorliegenden Grammatiken auf regulären Ausdrücken basieren, muss eine Neudefinition vorgenommen werden. Laut ANTLR-Dokumentation ist es üblich, das k nach der Länge des längsten Operators zu wählen. In Visual Basic 6 entspricht dies lediglich 2, weshalb das k entsprechend gewählt wurde.

Im Folgenden behandle ich die wichtigsten Besonderheiten der Lexer-Grammtik. Diese dienen dabei zum Teil als Grundlage für die spätere Beschreibung weiterer Aspekte der VB6-Analyse. Zudem beschreibe ich hier Manipulationen am Tokenstrom, der vom Lexer erzeugt wird.

4.1.3.1 Bezeichner und Schlüsselwörter

Der $LL(k)$-Ansatz macht die Definition von Schlüsselwörtern im Lexer schwierig. Problematisch sind die hier auftretenden gemeinsamen Präfixe zwischen den Anfangsbuchstaben der einzelnen Wörter. Dieses Problem ließe sich in $LL(k)$ nur durch ein recht groß gewähltes k lösen. ANTLR umgeht dies, indem die Schlüsselwörter nicht als Lexer-Regeln definiert werden. Stattdessen interpretiert der Generator alle in doppelten Anführungszeichen eingeschlossenen Literale in der Parsergrammatik (sowie die in einem speziellen Abschnitt der Gammatikdefinition angegebenen Literale) als Schlüsselwörter und nimmt diese in eine Liste auf. In der Produktion zur Erkennung von Bezeichnern kann der Lexer durch die Option `testLiterals` instruiert werden, jede durch die Regel erkannte Zeichenkette mit der Schlüsselwortliste abzugleichen. Sollte ein Schlüsselwort gefunden werden, wird ein entsprechender Token erzeugt.

Listing 4.1 zeigt eine Lexer-Produktion zur Erkennung von Bezeichnern und Schlüsselwörtern in einer simplen Form, die keine Sonderfälle behandelt. Hierbei beginnt die Regel mit ihrem Namen, gefolgt von den Optionen. Der Doppelpunkt leitet die Definition der Produktion ein, das Semikolon beendet die Produktion. Dazwischen folgt die Notation einer erweiterten BNF. Dabei sind andere Lexer-Produktionen generell in Großbuchstaben geschrieben.

Ein Bezeichner beginnt in Visual Basic 6 mit einem Buchstaben oder einem einzelnen Unterstrich, der von mindestens einem Buchstaben oder einer Ziffer gefolgt werden muss. Im weiteren Verlauf kann eine beliebige weitere Zahl von Buchstaben, Ziffern und Unterstrichen folgen.

```
IDENTIFIER
    options {testLiterals = true;} // Test identifiers for keywords
    :   (LETTER|('_')(LETTER|DIGIT)) (LETTER|DIGIT|'_')*
    ;
```

Listing 4.1: Stark vereinfachte Bezeichner-Produktion

Die Entscheidung, ob die Produktion ein Schlüsselwort oder einen Bezeichner erkannt hat, erfolgt erst im Nachhinein. Dies führt jedoch zu einem Problem mit den Kommentaren, die durch

das `Rem`-Schlüsselwort eingeleitet werden können. Es lässt sich keine Produktion definieren, die einen Kommentar von diesem Schlüsselwort abhängig macht, ohne eine Mehrdeutigkeit mit der `IDENTIFIER`-Produktion zu verursachen. Denkbar wäre eine Verdoppelung des k auf den Wert 4, um „Rem␣<Kommentartext>“ als Kommentar erkennen und von einer Variable wie „Rem123“ unterscheiden zu können. Allerdings wäre dies der einzige Fall, der ein solches k benötigt.

Hier bieten die syntaktischen Prädikate Abhilfe. Indem die Produktion um eine Alternative erweitert wird, die nur dann wählbar ist, wenn „Rem␣“ im Lookahead liegt, lässt sich dieser Sonderfall behandeln, ohne dass der Lookahead generell vergrößert werden muss. Listing 4.2 erweitert die Regel entsprechend.

```
IDENTIFIER
    options {testLiterals = true;} // Test identifiers for keywords
    :   ("rem" (' '|'\t')) => COMMENT_REM {_ttype = COMMENT;}
    |   (LETTER|('_')(LETTER|DIGIT)) (LETTER|DIGIT|'_')*
    ;
```

Listing 4.2: Bezeichner-Produktion mit Berücksichtigung von Rem-Kommentaren

Es gibt nun zwei Alternativen, wobei die erste nur dann vom Lexer gewählt wird, wenn das syntaktische Prädikat `("rem" (' '|'\t'))` erfüllt ist. Ist dies der Fall, so wird die Produktion `COMMENT_REM` aufgerufen, die die Form von `Rem`-Kommentaren spezifiziert, und abschließend der Typ des erzeugten Tokens zu `COMMENT` abgeändert (er wäre sonst `IDENTIFIER`). Dabei verursachen `COMMENT_REM` und `IDENTIFIER` keine Mehrdeutigkeit, da erstere Regel als `protected` definiert ist und daher nie vom Parser eigenständig als Alternative betrachtet wird. Sie kann nur explizit aufgerufen werden, wie es im obigen Beispiel der Fall ist.

Ein weiteres Problem bei der Erkennung von Bezeichnern stellen Sprungmarken dar. Sie sind nach den gleichen Regeln wie `IDENTIFIER` aufgebaut, müssen aber am Anfang einer Zeile (Spalte eins) beginnen und mit einem Doppelpunkt enden. Strenggenommen sind auch sie Bezeichner. Allerdings ist ihre Erkennung im Parser eine umständliche Angelegenheit, denn der Doppelpunkt ist in Visual Basic 6 zugleich der Operator, der mehrere Anweisungen auf einer Zeile konkateniert. Eine Sprungmarke liegt ausschließlich dann vor, wenn beide Bedingungen, also der Beginn in Spalte 1 und der Abschließende Doppelpunkt erfüllt sind. Zu allem Überfluss können Sprungmarken praktisch in jeder Zeile eingesetzt werden - selbst vor einem `End If`.

Um die Grammatikdefinition nicht unnötig zu verkomplizieren, sollten Sprungmarken daher schon im Lexer mit einem speziellen Tokentyp kenntlich gemacht werden. Ihre Behandlung im Parser wird dadurch deutlich vereinfacht. Sie können aber ebenfalls nicht in einer weiteren Produktion definiert werden, denn diese würde eine Mehrdeutigkeit mit `IDENTIFIER` hervorrufen. Sie lassen sich jedoch durch die Kombination eines semantisches Prädikates, das den Beginn in Spalte eins prüft, und eines syntaktischen Prädikats, das nach dem abschließenden Doppelpunkt Ausschau hält, innerhalb der `IDENTIFIER`-Produktion erkennen. Listing 4.3 erweitert die Produktion entsprechend.

```
IDENTIFIER
    options {testLiterals = true;} // Test identifiers for keywords
    :   {getColumn()==1}? ((LETTER|('_')(LETTER|DIGIT)) (LETTER|DIGIT|'_')* COLON) =>
            (LETTER|'_'(LETTER|DIGIT)) (LETTER|DIGIT|'_')* COLON {_ttype = LINE_LABEL
                ;}
    |   ("rem" (' '|'\t')) => COMMENT_REM {_ttype = COMMENT;}
    |   (LETTER|('_')(LETTER|DIGIT)) (LETTER|DIGIT|'_')*
    ;
```

Listing 4.3: Vollständige Bezeichner-Produktion

Die neue Erstalternative darf durch das semantische Prädikat (technisch die Bedingung eines `if`-Statements der generierten Sprache) nur dann gewählt werden, wenn der Lexer sich am Anfang einer Zeile befindet. In jedem anderen Fall wird sie vollkommen ignoriert. Das syntaktische Prädikat schaut über eine beliebige Länge von Token, nämlich so lange wie das angegebene Tokenmuster passt, voraus. Nur wenn beide Prädikate erfüllt sind, wird die Alternative durch den Parser geprüft und ein `LINE_LABEL`-Token erzeugt.

Die Entscheidung, ob ein Bezeichner oder ein Schlüsselwort vorliegt, bleibt hier offen. Wie ich zuvor schon beschrieben habe, ist es kontextabhängig, wann welche Schlüsselwörter reserviert sind. Der Lexer erzeugt für jedes Lexem, das mit einem der möglichen Schlüsselwörter identisch ist, stets den entsprechenden Schlüsselwort-Token. Es obliegt dem Parser die Unterscheidung vorzunehmen, da nur er auch die nötigen Kontextinformationen besitzt. Ich werde dieses Beispiel später in diesem Kapitel wieder aufgreifen und seine Lösung in der syntaktischen Analyse näher erläutern. Konsequenterweise werde ich mich auch bei der Beschreibung der semantischen Analyse auf Bezeichner konzentrieren, die eines der Hauptprobleme in der gesamten Arbeit darstellen.

4.1.3.2 Leerzeichen und Zeilenumbrüche

In vielen Sprachen dienen Leerzeichen und Zeilenumbrüche bei der lexikalischen Analyse lediglich dazu, die Grenzen einzelner Token zu ermitteln. Sie tragen in aller Regel keine Semantik in sich und sind auch für den Parser meist nicht relevant. Dort sind sie sogar eher hinderlich, da in den Produktionen stets auch die Leerzeichen vorgesehen werden müssten. Üblicherweise werden sie vom Lexer daher gar nicht erst in den Tokenstrom aufgenommen.

Die Zeilenumbrüche können in Visual Basic 6 jedoch nicht ignoriert werden, da sie zur Terminierung von Anweisungen dienen. Sie sind auch als Token an den Parser weiterzureichen und bei der syntaktischen Analyse relevant. Bei den Leerzeichen ist dies im Allgemeinen nicht der Fall, jedoch gibt es eine Ausnahme, die ich im Folgenden beschreibe.

With-Statements

Der Sonderfall hängt mit dem With-Statement zusammen. Diese Anweisung ermöglicht die Angabe einer Variable komplexen Typs, auf deren Member und Methoden innerhalb des With-Blocks zugegriffen werden kann, ohne dass die Variable dazu genannt werden muss. Listing 4.4 zeigt ein Beispiel für ein solches Statement.

```
With University.Students(i)      'liefert ein Student-Objekt zurueck
   .Semester = .Semester + 1     'inkrementiert das Attribut Semester im Student-Objekt
End With
```

Listing 4.4: Das With-Statement in VB6

Problematisch wird diese Syntaxform beim Aufruf von Prozeduren, deren Argumente, wie bereits beschrieben, nicht in Klammern gesetzt werden. Angenommen das Inkrementieren der Semesterzahl aus dem vorigen Beispiel würde von einer `Sub`-Prozedur namens `SemesterInc` erledigt, die eine Referenz auf die Semesterzahl als Parameter erhält, so sähe dies folgendermaßen aus: `SemesterInc .Semester`. Hierbei ist `.Semester` ein Prozedur-Argument. Werden die Leerzeichen nicht in den Tokenstrom aufgenommen, so sähe dieser wie in Abbildung 4.3 dargestellt aus.

Hieraus ist nicht mehr ersichtlich, ob `Semester` ein Attribut von einem Objekt ist, das `SemesterInc` referenziert oder das Argument einer Prozedur namens `SemesterInc`. Es gibt zwei Wege dieses Problem zu lösen:

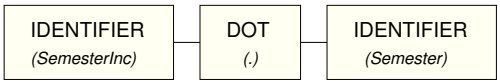

Abbildung 4.3: Tokenstrom ohne Leerzeichen

1. Aufgrund von Symbolinformationen, die Aufschluss darüber geben, was für eine Art von Symbol `SemesterInc` ist. Denn nach `Sub`-Prozeduraufrufen können nur Argumente und keine weiteren Qualifikationen folgen. Argumente von Funktionsaufrufen werden dagegen immer geklammert, Variablen können nicht von Argumenten gefolgt werden. Daher würde die Kenntnis des Symboltyps hier Klarheit schaffen.

2. Durch eine Repräsentationform im Tokenstrom, die den Sonderfall eindeutig ersichtlich macht.

Der erste Ansatz ist mit recht großem Zusatzaufwand im Parser verbunden, da das Symbol `SemesterInc` möglicherweise in einem Modul deklariert wird, das bislang noch gar nicht analysiert wurde. Wie ich im Rahmen der semantischen Analyse noch näher erklären werde, können Bezeichner nur dann eindeutig einem Symbol zugeordnet werden, wenn alle Symbole des Programms bekannt sind. Es müsste also zunächst eine vollständige Symboltabelle unter Betrachtung der gesamten Quelltexte generiert werden, bevor in einem zweiten Durchlauf ein korrekter Syntaxbaum erstellt werden kann.

Der zweite Lösungsweg ist dagegen einfacher umzusetzen. Zwar ist es keine Lösung alle Leerzeichen an den Parser zu übergeben, es ist aber möglich für das Zeichen „`.`“, also den Punkt, in diesem besonderen Fall einen speziellen Tokentyp `UNARY_DOT` zu erzeugen, den der Parser von den normalen Punkten unterscheiden kann. Konkret geht es also um die Unterscheidung eines binären und eines unären Qualifikationsoperators.

Im Lexer kann dies nicht passieren, da dieser wegen des $LL(k)$-Algorithmus keine Kenntnis der vorherigen Zeichen hat und demnach auch keine Entscheidungen von vorangehenden Leerzeichen abhängig machen kann. Die Produktion, die Leerzeichen erkennt, könnte zwar problemlos Punkte, die Leerzeichen folgen, erkennen, da diese sowieso im k von zwei lägen, nur reicht dies nicht, um alle unären Punktoperatoren zu erkennen. Diese können nämlich ebenso nach einer öffnenden Klammer (innerhalb einer Parameterliste) oder nach einem anderen unären Operator folgen (`+`,`-`,`^`). Wenn ein Konzept wie der unäre Punktoperator eingeführt wird, so sollte es auch konsequent angewendet werden. Die Unterscheidung beider Operatoren wird im Parser eine wichtige Rolle bei der Zusammensetzung von qualifizierten Bezeichnern spielen.

Die gewählte Lösung für das Problem liegt darin, Manipulationen auf dem vom Lexer generierten Tokenstrom durchzuführen. ANTLR bietet eigens zu diesem Zweck spezielle Interfaces an, die solche Eingriffe recht unkompliziert ermöglichen. Durch einen Tokenfilter, der den Typ des jeweils vorhergehenden Tokens speichert, kann so ein `DOT`-Token, das nach einem Leerzeichen, einer öffnenden Klammer oder einem unären Operator folgt, zu einem `UNARY_DOT`-Token umgeformt werden. Alle Leerzeichen können im gleichen Zuge aus dem Tokenstrom entfernt werden. Abbildung 4.4 zeigt dies an einem Beispiel.

Der Performanceverlust ist hierbei gering, da in die Tokenübergabe lediglich ein Funktionsaufruf und ein Switch über den Tokentyp, der ein numerisches Attribut ist, eingefügt wird. Die Entscheidung, dass ein Punkt in den gegebenen Situationen auch wirklich einen Qualifikationsoperator darstellt, ist möglich, da Punkte ansonsten nur in konstanten Zahlenwerten vorkommen, dort aber nicht am Beginn eines Wertes stehen dürfen. Ein Wert x für den $x < 1$ und $x > -1$ gelten, muss immer von einer Null angeführt werden.

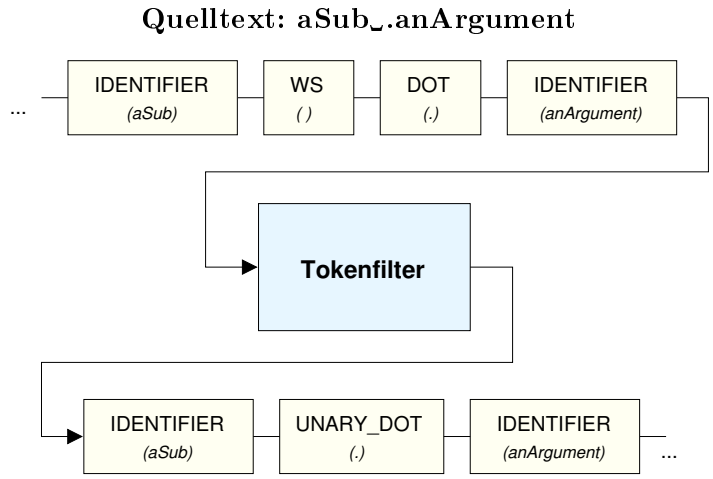

Abbildung 4.4: Markierung von unären Punkt-Operatoren durch einen Tokenfilter

Analog zum Problem des Punktes besteht die gleiche Problematik für den Bang-Operator (siehe 2.1.3). Dieser wird durch ein Ausrufungszeichen dargestellt und kann innerhalb eines With-Statements analog zum Punkt auch als unärer Operator auftreten. Daher wird der Bang-Operator auf die gleiche Weise in die Tokentypen `BANG` und `UNARY_BANG` eingeteilt.

4.1.3.3 For-Next-Schleifen

Eine weitere Anwendung für die Tokenfilterung ist die For-Next-Kontrollstruktur, die eine einfache Zählschleife darstellt. Üblicherweise wird jedem öffnenden `For` ein schließendes `Next` gegenübergestellt. Dem `Next` darf dabei der Bezeichner des Schleifenzählers angehangen werden. Enden mehrere solcher Schleifen an der gleichen Stelle im Quelltext, so können ihre `Next`-Enden zu einem zusammengefasst werden wie Listing 4.5 zeigt.

```
For i = 0 to 10
    For j = 0 to 10
        'do something
Next j, i
```

Listing 4.5: For-Next Schleifen mit vereintem Next

Für den Lexer stellt dies kein Problem dar, für den Parser hingegen schon. Zum einen wäre eine deutlich kompliziertere Produktion für diese undokumentierte Sonderform notwendig, zum anderen ist `Next` zwar ein Schlüsselwort, aber nicht reserviert. Demnach kann es auch eine Variable oder Prozedur dieses Namens geben und der Parser müsste `Next i, j` von einem Prozeduraufruf unterscheiden können. Durch Tests lässt sich herausfinden, dass Visual Basic ein `Next` offenbar immer dann als Schlüsselwort wahrnimmt, wenn es am Beginn einer Anweisung steht, von einer beliebigen Anzahl von kommagetrennten Bezeichnern gefolgt wird und die Anweisung nichts darüber hinaus enthält. Genau diese Bedingungen werdem durch einen Tokenfilter geprüft, der die Schleifenenden gegebenenfalls umformt. Aus `Next i, j` würde so `Next j : Next i`, wobei der Doppelpunkt in Visual Basic Trennzeichen für mehrere Anweisungen ist.

Darüber hinaus dienen die Tokenfilter der Vereinfachung einiger weiterer Tokenfolgen. Dies gilt vor allem für Konzepte, die durch mehrere Schlüsselwörter definiert werden. Beispiele hierfür sind `End If` oder `For Each`. Die zwei Token werden in jeweils einen Token umgeformt. Dies hilft im Parser nebenbei auch Mehrdeutigkeiten mit anderen Schlüsselwörtern zu ver-

meiden – `End` alleine beendet beispielsweise den Programmablauf. Insgesamt kommen zwei Filter zum Einsatz, wobei einer, wie bereits beschrieben, den jeweils letzten Token speichert, während der andere durch einen Puffer mit beliebiger Größe vorausschauen kann. Er führt also einen Lookahead durch. Diese Trennung dient vor allem einer besseren Übersichtlichkeit der Implementierung.

4.1.4 Syntaktische Analyse

Die syntaktische Analyse basiert auf der Grammatik, die nach dem zu Beginn dieses Kapitels beschriebenen Verfahren erstellt wurde. In diesem Abschnitt werde ich wie schon beim Lexer die wichtigen Aspekte Grammatik exemplarisch vorstellen. Dabei greife ich das Beispiel der Bezeichner und Schlüsselwörter wieder auf und erläutere darüber hinaus auch einige Besonderheiten, die in Visual Basic beim Aufruf von Prozeduren auftreten. Diese Beispiele dienen als Grundlage für den Abschnitt über die semantische Analyse. Zunächst werde ich aber auf den allgemeinen Aufbau der Grammatik näher eingehen.

Die Quelltexte der Module teilen sich in die zwei Bereiche der von Visual Studio verwalteten Kopfdaten und des benutzerdefinierten Programmcodes. Die Bestandteile des Modulkopfes sind je nach Art des Moduls unterschiedlich. Daher beinhaltet die Grammatik auch zum Teil modulspezifische Produktionen. Der benutzerdefinierte Teil des Quelltextes ist dagegen einheitlich. Zwar gibt es auch hier einige eigentlich modulspezifische Elemente – wie beispielsweise Events, die nicht in Standardmodulen deklariert werden dürfen. Allerdings ist es wie schon beschrieben nicht erforderlich derartige Einschränkungen in der Grammatik zu berücksichtigen, da davon ausgegangen wird, dass nur korrekte Programme analysiert werden. Daher wird der benutzerdefinierte Bereich durch vereinheitlichte Produktionen beschrieben. Dies hat den Vorteil, dass die Grammatikdefinition die Syntax nicht bis ins kleinste Detail beschreiben muss und so kompakter und übersichtlicher gehalten werden kann. Hierzu gehört auch, dass die VB6-Konvention in Modulen alle Deklarationen vor den Prozeduren anzugeben, nicht explizit nachgebildet werden muss.

Bei den Kopfdaten funktioniert diese vereinheitlichte Sichtweise nicht Die Syntax der unterschiedlichen Modultypen ist zu unterschiedlich. Zudem werden diese Unterschiede genutzt, um die Art des Moduls anhand des Quelltextes zu erkennen. Sie müssen also auch explizit in der Grammatik nachgebildet werden. Abbildung 4.5 listet die verschiedenen Bereiche und die Module, in denen sie auftreten können, auf. Dabei entspricht die Reihenfolge ihrem Auftreten in den Modulen.

Während die Syntax der Modulköpfe relativ simpel ist, bedeutete die Definition der Grammatik für den benutzerdefinierten Programmanteil einen wesentlich größeren Aufwand. Hier mussten die Vielfalt der Sprache und ihre syntaktischen Besonderheiten berücksichtigt werden. Die allgemeine Struktur der Grammatik ist dabei nicht unüblich. Auf der Modulebene werden zunächst Deklarationen und Prozeduren erwartet, auf der Prozedurebene dann Anweisungen, die wiederum Ausdrücke enthalten können, die rekursiv über ihre Operatoren definiert sind, die wiederum Bezeichner und konstante Werte miteinander verknüpfen. Die Schwierigkeiten machen sich in Details wie den sehr liberalen Konventionen für Bezeichner und Schlüsselwörter und einzelnen syntaktischen Besonderheiten bemerkbar, die ich mit den später in diesem Abschnitt folgenden Beispielen beschreiben werde.

Das Ergebnis der syntaktischen Analyse sind abstrakte Syntaxbäume, die alle für die semantische Analyse benötigten Teile des Quelltextes enthalten. Token, die nur syntaktische Bewandtnis haben oder durch die Baumstruktur obsolet werden, wie beispielsweise Zeilenumbrüche, sind dort nicht mehr enthalten. Zudem sind die Bäume nur soweit strukturiert, wie

Quelltextbereich	Modulart		
	Standard	Klasse	Formular
Modulversion und –typ		X	X
Einbindung von COM–Objekten			X
Initialisierung visuell erstellter Komponenten		X	
Klassenoptionen			X
allgemeine Modulattribute und –optionen	X	X	X
benutzerdefinierter Quelltext	X	X	X

Abbildung 4.5: Quelltextbereiche nach Modulen aufgeschlüsselt

es die weiteren Analysen erfordern. Beispielsweise werden komplexere Anweisungen, deren Semantik nicht weiter von Bedeutung ist, nicht in Baumstrukturen überführt, sondern bilden für sich betrachtete eine verkettete Liste. Dies ist die Struktur, die ANTLR standardmäßig generiert. Um einen Baum zu erhalten, muss die Generierung durch spezielle Annotationen in den Produktionen gesteuert werden. Dies habe ich nur dort vorgenommen, wo eine Struktur vonnöten ist. So bilden beispielsweise alle Anweisungen einer Prozedur einen gemeinsamen Teilbaum.

Die von ANTLR generierten Bäume bestehen nur aus Knoten einer einzigen Knotenklasse. Zur Unterstützung der semantischen Analyse werden an einigen Stellen jedoch besondere Knotentypen mit erweiterter Funktionalität eingesetzt. Ihre Beschreibung soll nicht an dieser Stelle stattfinden, da sich ihr Sinn erst bei der semantischen Analyse offenbart. Daher werde ich dort näher auf die Erweiterung von ANTLRs Baumklassen und die Datenstrukturen im allgemeinen eingehen. Dieser Abschnitt konzentriert sich allein auf die Grammatik.

Der erzeugte Parser kommt, wie auch schon der Lexer, mit einem k von 2 aus, wobei an einigen Stellen syntaktische Prädikate verwendet werden, um schwierige Konstrukte mit einem erweiterten Lookahead zu erkennen. Im Folgenden werde ich die oben bereits erwähnten Beispiele aus der Grammatik vorstellen.

4.1.4.1 Unterscheidung von Bezeichnern und Schlüsselwörtern

Die Unterscheidung von Bezeichnern und Schlüsselwörtern wird im Parser auf Grundlage der Grammatikdefinition vorgenommen. Der Lexer kann dies, wie beschrieben, wegen der fehlenden Kontextinformationen nicht leisten. Stattdessen erzeugt er immer ein Schlüsselwort-Token, wenn er auf ein Lexem stößt, das ein Schlüsselwort sein könnte. Das bedeutet, dass Bezeichner, deren Namen sich mit Schlüsselwörtern überschneiden, im Parser nicht als `IDENTIFIER`-Token sondern fälschlich als Schlüsselwort-Token eingehen.

Schon mehrfach habe ich erwähnt, dass die Unterscheidung von Bezeichnern und Schlüsselwörtern bei der Analyse von VB6-Quelltexten problematisch ist. Der Grund dafür liegt darin, dass diese vom Kontext, in dem das Wort verwendet wird, abhängt und die Regeln sehr liberal sind. Die Mehrzahl aller Schlüsselwörter kann in verschiedenen Situationen auch als Bezeichner dienen, wobei es sich als schwierig erweist, die Regeln nach denen sich dies entscheidet vollständig nachzuvollziehen. Ich wiederhole das Beispiel hierzu aus Abschnitt 2.1 in

Listing 4.6, um die Problematik daran näher zu erläutern.

```
Sub Sub()                 'Compilerfehler
   'Anweisungen...
End Sub

Type myType
   Sub As String          'erlaubt
End Type

Property Get Name()
   'Anweisungen...
End Property

Sub Property()            'erlaubt
   'Anweisungen...
End Sub

Sub anotherSub()
   Property               'Compilerfehler
   Call Property          'erlaubt
End Sub
```

Listing 4.6: Kontextabhängige Reservierung von Schlüsselwörtern

Besonders das letzte Beispiel, in dem die zuvor deklarierte `Sub`-Prozedur namens `Property` nicht direkt aufgerufen werden kann (unter Verwendung des `Call`-Statements hingegen schon) verdeutlicht, wie schwer es fällt, hier Regelmäßigkeiten zu erkennen. Syntaktisch spräche nichts dagegen, das Wort `Property` innerhalb einer Prozedur als Bezeichner zu verwenden, da sich keine Mehrdeutigkeiten mit anderen Teilen der Syntax ergäben. Ebenso ist nicht verständlich warum eine `Sub`-Prozedur den Namen `Property` tragen darf aber nicht den Namen `Sub`. Die Signatur `Sub Sub()` lässt sich syntaktisch recht einfach erkennen, sofern `Sub` auch ein Bezeichner sein darf.

Im Rahmen dieser Arbeit konnte ich kein Regelwerk finden, das diese Sonderfälle vollständig erklärt. Es entsteht eher der Eindruck, dass Visual Basic hier möglicherweise gar keinem generellen Prinzip folgt, sondern die Reservierung von Schlüsselwörtern zumindest teilweise vom konkreten Fall abhängig macht.

Wie schon zuvor bemerkt ist es aber gar nicht notwendig, derartige Einschränkungen im Detail nachzubilden. Die Grammatik darf auch hier stark verallgemeinern, da die Eingabeprogramme stets korrekt sind. Wichtig sind lediglich zwei Dinge: Die vorgenommene Einteilung muss zu korrekten Klassifizierungen führen und die Vereinfachung der Grammatik sollte möglichst keine Mehrdeutigkeiten zur Folge haben. Bei der testweisen Verfeinerung offenbart sich schnell eine Erkenntnis, die auch in der JavaCC-Grammatik zur Lösung dieses Problems verwendet wurde: In der Definition der Membervariablen von VB6-Types scheinen beinahe alle Regeln außer Kraft gesetzt zu sein. Ebenso können nach dem Bang-Operator nahezu beliebige Bezeichner verwendet werden, da es sich hierbei um die Schlüssel von Mengenelementen handelt (vgl. 2.1.3). Die Regeln der sonstigen Bezeichner sind strikter, scheinen sich aber recht ähnlich.

Dies legt nahe, zwei Bezeichnerregeln einzuführen, die jeweils eine unterschiedliche Menge von Schlüsselwörtern einschließen. Je nach Kontext kann dann in der Grammatik die Bezeichnerproduktion mit der benötigten Restriktionsstufe aufgerufen werden. Mit dieser Einteilung ist es möglich, alle Beispielprogramme erfolgeich zu analysieren, ohne dabei neue Mehrdeutigkeiten in der Grammatik zu verursachen. Die striktere beider Regeln nennt sich `var_identfier`, die liberalere heißt `member_identifier`. Beide akzeptieren `IDENTIFIER`-Token, sowie eine fest vorgegebene Menge von Schlüsselwort-Token. Dabei definiert `member_identifier` eine echte Obermenge von `var_identifier`. Listing 4.9 zeigt ein gekürztes Beispiel für diese Produktionen und ihre Anwendung. Die Syntax ist dabei mit jener der Lexerproduktionen identisch.

Neu sind hier lediglich die kleingeschriebenen Produktionsnamen. Schlüsselwörter sind nicht als Token, sondern, wie in ANTLR üblich, in ihrer (case-insensitiven) Textform angegeben.

```
sub
    :    "sub" var_identifier LPAREN argument_list RPAREN EOL
         ...
    ;

type
    :    "type" var_identifier EOL
             ( member_identifier ( "as" type_spec )? )+
         "end" "type"
    ;

var_identifier
    :    IDENTIFIER
    |    "error"
    |    "property"
    |    ...
    ;

member_identifier
    :    var_identifier
    |    "sub"
    |    "function"
    |    ...
    ;
```

Listing 4.7: Produktionen für Sub-Prozeduren und Bezeichner (Ausschnitt)

Beide Bezeichnerproduktionen ändern den Typ des erzeugten Baumknotens für als Bezeichner erkannte Schlüsselworttoken in `IDENTIFIER` um, so dass im erzeugten Syntaxbaum die korrekte Klassifizierung von Bezeichnern und Schlüsselwörtern enthalten ist. Die Auswahl welche Schlüsselwörter in welche dieser Produktionen aufzunehmen sind, ist ein Ergebnis des Versuch-und-Irrtum-Prinzips. Die Untersuchung weiterer Quelltexte könnte also im Besonderen an dieser Stelle weitere Verfeinerungen erfordern. Der zeitliche Rahmen dieser Arbeit und die begrenzte Menge an Testfällen haben vermutlich keine vollständigen Listen hervorgebracht. Diese ließen sich nachträglich jedoch noch ermitteln, indem ein automatisierter Test mit allen Schlüsselwörtern in der Grammatik und allen Einsatzmöglichkeiten in Deklarationen durchgeführt wird. Hierzu könnten Testfälle automatisch generiert werden. Ein ähnliches Verfahren habe ich bei der semantischen Analyse erfolgreich eingesetzt, um Details der Namensregeln in Visual Basic 6 in Erfahrung zu bringen.

An einigen Stellen führt die Überschneidung von Schlüsselwörtern und Bezeichnern jedoch zu unvermeidlichen Mehrdeutigkeiten in der Grammatik, die lokal gelöst werden müssen. Ein Beispiel hierfür ist die Spezifikation von Prozedur-Parametern. Laut [Mic06b] folgt diese dem Muster aus Listing 4.8. Hierbei sind Elemente in eckigen Klammern optional.

```
[Optional] [ByVal | ByRef] [ParamArray] varname[( )] [As type] [= defaultvalue]
```

Listing 4.8: Offizielle Spezifikation für Prozedurparameter aus [Mic06b]

Der Name des Parameters, hier `varname` genannt, darf „Optional“ lauten, aber auch „As“. Somit kann `Optional As Integer` mit dem fixen k von 2 nicht erkannt werden. Befinden sich die Lexeme `Optional As` im Lookahead, so kann ein optionaler Parameter namens `As` folgen (`Optional As As Integer`), genauso aber ein Parameter Namens `Optional` mit folgender Typangabe (`Optional As Integer`). Zur sicheren Unterscheidung wäre hier ein k von mindestens 4 nötig. Solche Probleme lassen sich jedoch auch ohne eine Erhöhung des allgemeinen k durch die Verwendung von syntaktischen Prädikaten lösen, indem der Lookahead lokal entsprechend erweitert wird.

Generell stellt sich bei solchen Fällen die Frage, ob ein größeres k, das Performanceeinbußen bedeutet, oder ein syntaktisches Prädikat, das der Lesbarkeit der Grammatik abträglich ist, zur Lösung eingesetzt werden sollte. Die Grammatik verwendet in vierzehn Fällen syntaktische Prädikate, allerdings würden nur drei davon durch die Erhöhung des k auf den Wert 3 verzichtbar. Acht der verwendeten syntaktischen Prädikate lassen sich gar nicht durch die Erhöhung des k ersetzen, da sie einen beliebig langen, weil rekursiven, Lookahead definieren. Ich habe daher entschieden, das k bei 2 zu belassen und dafür eine überschaubare Anzahl von syntaktischen Prädikaten hinzunehmen.

4.1.4.2 Qualifizierte Bezeichner

Die Unterscheidung von `var_identifier` und `member_identifier` muss auch bei der Verwendung von Bezeichnern in den Ausdrücken des Quelltextes beachtet werden. Während eine Qualifizierung stets mit einem `var_identifier` beginnt, können im weiteren Verlauf auch `member_identifier` folgen, da die Member eines komplexen Typs referenziert werden können. Ebenso ist es möglich, dass die bereits erwähnten Bang-Operatoren zum Einsatz kommen. Hier folgt dem Operator der Schlüssel eines Collection-Elements. Folglich kann hier ein beliebiger Bezeichner – auch ein Schlüsselwort – stehen. Auch dieser Fall wird mit dem `member_identifier` abgedeckt. Listing 4.9 zeigt die Produktionen, die Bezeichner in Ausdrücken der Sprache definieren. Hier ist auch die Verwendung der unären Qualifikationsoperatoren, die ich in der Beschreibung der lexikalischen Analyse erwähnt habe, zu sehen.

```
id
    :   var_identifier ( identifier_suffix )*
    |   ( UNARY_DOT | UNARY_BANG ) member_identifier ( identifier_suffix )*
    ;

identifier_suffix
    :   ( DOT | BANG ) member_identifier
    |   bracketed_argument_list
    ;

bracketed_argument_list
    :   LPAREN argument_list RPAREN  //argument_list darf leer sein
    ;
```

Listing 4.9: Produktionen zur Erkennung von Bezeichnern (vereinfacht)

Die Tatsache, dass einem Bezeichner mehrere geklammerte Argumentenlisten folgen dürfen, entspricht den Besonderheiten, die ich bereits in Abschnitt 2.1.3 definiert habe und ist in diesem Fall keine Ungenauigkeit. Zur besseren Verständlichkeit fasse ich die Klammerungen, die Visual Basic erlaubt, noch einmal zusammen. Es handelt sich dabei um die folgenden drei Formen:

- **Funktionsargumente**

  Der Standardfall ist die Angabe einer Liste von Argumenten für einen Funktionsaufruf. Dies kann nur direkt nach einem Bezeichner geschehen. Während Argumente generell geklammert werden müssen, kann bei parameterlosen Funktionen jedoch auf die Klammerung verzichtet werden. Zudem werden die Argumente von `Sub`-Prozeduren nicht geklammert, sofern nicht die `Call`-Anweisung dem Aufruf vorangestellt wurde.

- **Aufruf von Standardeigenschaften**

  Klassen können eine einzelne Eigenschaft zu ihrer Standardeigenschaft erheben. Diese ist dann nicht nur explizit durch die Angabe ihre Namens aufrufbar, sondern auch

über eine spezielle Kurzform. Hierbei kann die Argumentenliste für den Aufruf der Standardeigenschaft direkt an den Bezeichner einer Objektinstanz angehängt werden. Auf die gleiche Weise funktioniert der Dictionary Lookup – `fruits(banana)` ist identisch mit `fruits.Item(banana)`, wenn `fruits` eine Collection ist. Solch ein Aufruf ist auch dann erlaubt, wenn das vorige Element in einer Qualifizierung ein Objekt mit Standardmethode zurückliefert. Angenommen `fruits` ist selbst Teil eine Collection `shoppingList`, so könnte `shoppingList(fruits)(banana)` ein korrekter Bezeichner sein, wobei hier zwei Mal eine Standardeigenschaft beziehungsweise die Dictionary Lookup Methode (meist `Item`) aufgerufen wird.

- **Zugriff auf Arrayfeld**

  Auch auf Arrayfelder wird zugegriffen, indem runde Klammern verwendet werden. Daher ist dieser Ausdrucksform mit einem Funktions- oder Standardeigenschafts-Aufruf mit einem Argument identisch. Auch er kann hinter einer Klammerung auftreten, wenn diese ein Array zurückgeliefert hat.

Somit lassen sich diese drei semantisch verschiedenen Klammerungen syntaktisch nicht zuverlässig unterscheiden. Dies ist aber bedeutsam für die RFG-Generierung, da die Aufrufe von Standardeigenschaften nur anhand der Klammerung den Aufruf einer Property-Prozedur kennzeichnen. Hier findet also gewissermaßen ein anonymer Aufruf statt, der allerdings früh gebunden wird. Da sich die Art einer Klammerung erst später mit Hilfe von Symbolinformationen auflösen lässt, erzeugt der Parser einen Teilbaum, der von einem allgemeinen Knoten vom Typ `BRACKETED_ARGUMENT_LIST` als Wurzel angeführt wird.

4.1.4.3 Prozedurargumente

Ein größeres Problem ergibt sich durch die spezielle Syntax von `Sub`-Prozedur-Aufrufen. Ihre Argumente werden in der Regel nicht in Klammern gesetzt, was gleichzeitig zu einem gemeinsamen Präfix mit der geklammerten Argumentenliste führt und zu Mehrdeutigkeiten innerhalb von Ausdrücken. Das gemeinsame Präfix ergibt sich dadurch, dass ein Argument in Form eines eingeklammerten Ausdrucks angegeben werden kann. Dieser ist syntaktisch identisch mit einer geklammerten Argumentenliste, die nur aus einem Argument besteht. Listing 4.10 zeigt einen solchen Fall.

```
eineFunktion    (1 + 2 - eineFunkiton(x))
eineSubProzedur (1 + 2 - eineFunkiton(x)), 2
                                        '^ Sub-Aufruf erst ab dem Komma ersichtlich
```

Listing 4.10: Erst spät erkennbare ungeklammerte Argumentenliste

Eigentlich sollte dies ein Ausnahmefall sein, da die Klammerung eines kompletten Arguments auf den ersten Blick vollkommen bedeutungslos erscheint. In Visual Basic ist sie jedoch mit einer besonderen Bedeutung verbunden: Geklammerte Argumente werden stets als *Wert* übergeben, auch wenn der entsprechende Parameter als *Referenzparameter* definiert wurde. Daher ist die Verwendung von Klammerungen auch nicht unüblich sondern wird zur Steuerung des Übergabeverhaltens benutzt. Zwar ist dies in Bezug auf den RFG ein unwesentliches Detail, da dort das Übergabeverhalten nicht abgebildet wird, im Sinne einer späteren Nutzung des Parsers für weitere Zwecke sollte dies aber von Beginn an mit Bedacht behandelt werden.

In jedem Fall sollte eine Unterscheidung zwischen beiden Fällen stattfinden, damit der erste Aufruf aus dem Beispiel als geklammerte Argumentenliste und der zweite als ungeklammerte Argumentenliste mit einem geklammerten Ausdruck als erstes Argument erkannt wird. *Left*

Factoring, wie es in [ASU86, S. 178] beschrieben wird, bietet hier keine sinnvolle Lösung, da es sich bei beiden Ausprägungen der gleichen Syntax um semantisch unterschiedliche Konstruktionen handelt, die dann aber durch den gleichen Knoten im AST repräsentiert würden – ganz abgesehen davon, dass die Verständlichkeit der Grammatik durch eine solche Aufspaltung leiden würde.

Das andere Problem ergibt sich dadurch, dass ein Prozedur-Argument selbst einen Ausdruck darstellt. Wenn es aber nicht durch Klammern abgesetzt wird, so führt es zu Mehrdeutigkeiten mit anderen Ausdrücken, in die ein Prozeduraufruf zumindest theoretisch eingebettet sein könnte, ließe man ihn dort zu. Das heißt, wenn die zuvor gezeigte Bezeichnerregel um eine optionale ungeklammerte Argumentenliste erweitert würde, so entstünden Mehrdeutigkeiten in der Grammatik. Und zwar in den Produktionen für die Operatoren in Ausdrücken. Das Problem lässt sich an einem Beispiel verdeutlichen:

- Bezeichner, qualifizierte Bezeichner sowie konstante Werte sind Ausdrücke.
- `<Ausdruck> + <Ausdruck>` ist ebenfalls ein Ausdruck.
- Ist `S` eine `Sub` mit einem Parameter dann ist `S 1` ein Aufruf dieser Prozedur und ebenfalls ein Ausdruck.

  ⇒ `S 1 + 2` wäre auch ein korrekter Ausdruck, wobei unklar ist, ob `+ 2` dem Argument zugehörig ist oder nicht. Der Parser hätte hier keinen Anhaltspunkt, um festzustellen, wo ein Ausdruck endet und der nächste beginnt. Bei einer Klammerung der Argumente wäre dies kein Problem, da der Ausdruck `S(1) + 2` lautete und frei von Mehrdeutigkeiten wäre (Ein $LL(k)$-Parser erkennt hier die Klammerung für die Argumente, da sie auf einen Bezeichner folgt. Er kann am Ende der Klammerung erkennen, dass der Aufruf endet und der Ausdruck fortgesetzt wird).

In der Praxis kann dieser Fall eigentlich gar nicht auftreten, denn `Sub`-Prozeduren liefern keinen Rückgabewert und dürfen daher auch nicht dort stehen, wo ein Wert benötigt wird – also auch nicht in einem solchen Ausdruck. Tatsächlich dürfen solche Aufrufe nur in Form einer abgeschlossenen Anweisung, die neben dem `Sub`-Aufruf weiter nichts enthält, vorkommen. Die Mehrdeutigkeiten entstehen in der Grammatik allerdings in jedem Fall, wenn man die ungeklammerte Argumenteliste zwischen Ausdrücken zulässt. Hier liegt das Problem: ANTLR erzeugt in diesem Fall eine Vielzahl von Warnungen wegen eben dieser Merhdeutigkeiten. Diese Warnungen könnten zwar ignoriert werden, allerdings bergen sie die Gefahr, dass sie mit Warnungen, die wegen anderer Mehrdeutigkeiten entstehen, zusammenfallen und die zusätzlichen Probleme so verdecken. Gerade bei der testbasierten Grammatikerstellung sind die Warnungen ein wichtiges Hilfsmittel um Fehler zu erkennen, die dem komplexen Regelwerk der Grammatik mit bloßem Auge nicht anzusehen sind.

Zusammenfassend lässt sich feststellen, dass es mit unverhältnismäßigen Nachteilen bezüglich der Verständlichkeit der Grammatik und des gesamten Rekonstruktionsprozesses verbunden wäre, `Sub`-Aufrufe ebenfalls in den schon bestehenden Produktionen für Bezeichner abzubilden. Es gibt jedoch einen Kompromiss, der lediglich die Einführung einer gewissen Redundanz in der Grammatik erfordert:

Da `Sub`-Prozeduren nur in einer eigenen abgeschossenen Anweisung aufgerufen werden können, kann eine weitere Produktion hinzugefügt werden, die den Sonderfall einer Anweisung, die nur aus einem (eventuell qualifizierten) Bezeichner besteht, abdeckt. Hier kann die Unterscheidung der Klammerungen durch ein syntaktisches Prädikat durchgeführt und die ungeklam-

merte Argumentenliste am Ende hinzugefügt werden, ohne dass Mehrdeutigkeiten entstehen. Listing 4.11 definiert eine solche Produktion.

```
simple_call_statement
   :   ( "call" )?
       (    var_identifier
       |    ( UNARY_DOT | UNARY_BANG ) member_identifier
       )

       (   ( DOT | BANG ) member_identifier
       |   (bracketed_argument_list (eos|DOT|BANG|"else"|bracketed_argument_list)) =>
              bracketed_argument_list
       )*

       ( argument_list )?
   ;
```

Listing 4.11: Anweisungen aus nur einem (evtl. qualifizierten) Bezeichner (vereinfacht)

Dies führt allerdings zu einer Mehrdeutigkeit mit dem anderen Fall, in dem ein Bezeichner am Anfang einer Anweisung stehen kann: der Zuweisung. Anstatt dies durch Left Factoring zu lösen, wird der Zuweisungs-Produktion kurzerhand ein syntaktisches Prädikat vorangestellt, das über den Bezeichner hinweg schaut und dort nach einem Zuweisungsoperator sucht. So lassen sich beide Fälle auf einfachem Wege unterscheiden. Auf den ersten Blick erscheint es sinnvoller, obige `simple_call_statement`-Produktion einfach so zu erweitern, dass sie anstatt der Argumentenliste den Zweisungsoperator gefolgt von einem Ausdruck erlaubt. Leider ist der Zuweisungsoperator (`=`) mit dem Vergleichsoperator identisch, wodurch hier wieder eine Mehrdeutigkeit entstünde.

Diese Lösung mag nicht sonderlich elegant sein, erfüllt jedoch ihren Zweck ohne die Grammatikdefinition unverhältnismäßig zu verkomplizieren. Die syntaktischen Prädikate sind sehr einfach und die Produktionen noch immer einigermaßen lesbar. Ihr einziger Nachteil besteht darin, dass einzelne `Sub`-Argumente, die in Form eines geklammerten Ausdrucks angegeben sind, als geklammerte Argumentenliste missinterpretiert werden. Für die RFG-Generierung ist das hinnehmbar. Sollte der Parser später für Zwecke eingesetzt werden, die eine Unterscheidung des Übergabeverhaltens erfordern, so sollten diese Fehler in der semantischen Analyse anhand der Symboltabelle erkannt und korrigiert werden.

In der Praxis reicht aber auch diese Lösung noch immer nicht aus, denn Visual Basic 6 enthält fünf intrinsische `Sub`-Prozeduren, die ihre ganz eigene Syntax zur Angabe von Argumenten haben. Die nähere Beschreibung dieser Ausreißer, die vornehmlich dem Zeichnen auf Oberflächenkomponenten dienen, soll hier erspart bleiben. Sie können in der Referenz [Mic06b] unter `Circle`, `Line`, `PSet`, `Scale` und `Print` nachgeschlagen werden (dabei sei angemerkt, dass dort die Syntax für `Line` zu allem Überfluss falsch angegeben ist). Ihre Syntax lässt sich nicht ohne weiteres mit der Definition für allgemeine Argumentenlisten zusammenführen, weshalb speziell für diese Fälle eigene Argumentenmuster definiert wurden. Zur Erkennung, welche Art von Prozeduraufruf vorliegt, wird, wie schon zuvor bei den geklammerten Argumentenlisten, Backtracking durch syntaktische Prädikate betrieben. Der Parser versucht dabei zunächst vorausschauend eine gewöhnliche Argumentenliste zu erkennen und wählt nur im Erfolgsfall tatsächlich auch die entsprechende Produktion `argument_list`. Scheitert der Lookahead dagegen, werden die folgenden Alternativen getestet, die die Sonderformen beinhalten. Listing 4.12 zeigt den entsprechenden Ausschnitt der Produktion. Hierbei stehen `drawing_method_call` und `print_method_call` für die beiden Produktionen, die die eigenwillige Syntax der Prozedur-Argumente erkennen.

```
(    (argument_list) => argument_list
|    (drawing_method_call) => drawing_method_call
|    print_method_call
)?
```

Listing 4.12: Produktion zur Erkennung unterschiedlicher Prozeduraufrufe (Ausschnitt)

Zusammenfassend wurde also der Aufruf einer Prozedur zusätzlich als einzelne Anweisung definiert, um auch die ungeklammerte Argumentenliste zu erlauben ohne gleichzeitig Mehrdeutigkeiten in der Grammatik zu verursachen. `Sub`-Aufrufe sind strenggenommen keine Ausdrücke mehr. Um dies zu ermöglichen musste die Produktion für Zuweisungen zusätzlich um ein syntaktisches Prädikat erweitert werden. Diese detaillierten Beispiele zeigen, dass die Grammatikdefinition nicht nur durch die reine Vielfalt an Anweisungen und Schlüsselwörtern sondern auch aufgrund der allgemein wenig strikten und uneinheitlichen Syntax und den liberalen Regeln verkompliziert wird.

4.2 Analyse von COM-Bibliotheken

In Abschnitt 2.2 habe ich das Component Object Model beschrieben. Dort ergab sich die Anforderung, dass die in den COM-Komponenten binär gespeicherten Schnittstellendefinitionen ausgelesen und in Format gebracht werden müssen, das sich einfach auf das System, auf dem die Analyse stattfinden soll, übertragen lässt. Dies ist logischerweise nur für solche COM-Komponenten möglich, die ihre Schnittstellendefinitionen verfügbar machen, sprich die eine so genannte *Typelib* enthalten. Es ist anzumerken, dass solche Typbibliotheken auch losgelöst von der COM-Komponente existieren können. Die systemeigenen Prozeduren und Klassen von Visual Basic 6 liegen beispielsweise in Form einer reinen Typelib vor. Daher werden diese Symbole ebenfalls über die von mir „COM-Analyse“ genannte Komponente gewonnen.

Es muss ein Weg gefunden werden, diese Informationen auszulesen und ein Format für ihre Übertragung festgelegt werden. Dieses Format muss selbst wiederum ausgelesen werden, damit die Bibliothekssysmbole in die semantische Analyse einbezogen werden können. Ich betrachte zunächst die Möglichkeiten zum Auslesen der Typbibliotheken, da sich hieraus Konsequenzen für die Wahl des Formats ergeben.

4.2.1 Auslesen von COM-Bibliotheken

Microsoft verwendet eine erweiterte Form der ***I**nterface **D**efinition **L**anguage* (IDL)[5], die den Namen *MIDL*[6] trägt, um es Programmierern zu ermöglichen, Typbibliotheken zu generieren. Ein gleichnamiger Compiler übesetzt die MIDL-Spezifikationen in das binäre Typelib-Format. Auch der umgekehrte Weg ist mit dem Windows-Entwicklungswerkzeug *Oleview* möglich. Damit lassen sich MIDL-Spezifikationen für Typbibliotheken generieren, was allerdings einer manuellen Behandlung jeder einzelnen Bibliothek bedarf und daher für Projekte, die viele COM-Bibliotheken verwenden, kaum praktikabel ist. Zudem sind Microsofts Modifikationen an der IDL ebenfalls nicht formal in Form einer vollständigen Grammatik beschrieben. Auch hier gibt es nur eine Onlinereferenz zu den einzelnen Schlüsselwörtern. Es gibt zwar eine fertige IDL-Grammatik für ANTLR, allerdings ist diese nicht in der Lage, die MIDL-Spezifikationen korrekt zu erkennen, sondern bezieht sich auf den Standard der OMG.

[5] IDL: `http://www.omg.org/cgi-bin/doc?formal/02-06-39`

[6] MIDL: `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/midl/midl/midl_start_page.asp`

Eine andere Möglichkeit zum Inspizieren von Typelibs liegt in der COM-Bibliothek *Tlbinf32*, die Funktionen bereitstellt, mit denen sich die Schnittstellen anderer Bibliotheken abfragen lassen. Die Dokumentation zu dieser Bibliothek enthält eine Beispielfunktion, die die Schnittstellen einer Bibliothek in Signaturen der VB6-Syntax transformiert. Hierbei werden Konzepte, die VB6 nicht kennt - wie beispielsweise Interfaces - bereits auf entsprechende VB6-Konstrukte abgebildet. Interfaces werden beispielsweise als Klassen dargestellt. Die formale Beschreibung solcher Signaturen liegt mit der Erstellung der VB6-Grammatik bereits weitestgehend vor.

Daher ist die Verwendung der Tlbinf32-Bibliothek der Analyse der MIDL vorzuziehen. Für letztere liegt keine vollständige Grammatik vor und die Generierung von MIDL-Definitionen ist ein sehr umständliches Unterfangen. Die Tlbinf32-Bibliothek kann dagegen durch den vorliegenden Beispielcode bereits VB6-Signaturen erzeugen und lässt sich zudem mit relativ wenig Aufwand erweitern, um gezielt Bibliotheken auszulesen.

Die Analyse wird von einem Windows-Kommandozeilenprogramm namens *COM2VB6* durchgeführt, das wahlweise dazu verwendet werden kann, Bibliotheken anhand ihrer Dateinamen oder GUID zu untersuchen. Die Information, welche Bibliotheken von einem VB6-Projekt eingebunden werden, ist in einer zentralen Projektdatei gespeichert. Alle COM-Komponenten sind anhand ihrer GUID referenziert. Um die Analyse eines Projektes möglichst einfach zu gestalten, wurde das Analysewerkzeug so konzipiert, dass es anhand der Projektdatei die benötigten COM-Komponenten ermittelt und über ihre GUID und die Windows-Registry anspricht. Dies erfordert jedoch auch, dass alle Komponenten auf dem Rechner installiert und in der Registry angemeldet sind. Dies ist auf dem Rechner, auf dem das zu untersuchende Softwareprojekt entwickelt wurde, per Definition der Fall.

Das Ergebnis der Analyse sind menschenlesbare Textdateien, deren Name aus der GUID der COM-Komponente besteht. Sie können später anhand dieses Schlüssels referenziert werden. Diese Dateien können anhand eines einfachen Signatur-Parsers, der mittels ANTLR eigens hierfür erstellt wurde, ausgelesen werden. Seine Grammatik konnte direkt aus der VB6-Grammatik abgeleitet werden, da diese bereits sehr ähnliche Signaturen erkennen kann. Der Parser erzeugt jedoch keine Bäume, sondern liefert gleich die benötigten Einträge für die Symboltabelle, die ich später genauer beschreiben werde. Somit können die Daten für die semantische Analyse bereitgestellt werden. Abbildung 4.6 zeigt die Zusammenhänge in der COM-Analyse.

4.2.2 Zwischenformat zur Übertragung der Schnittstellendefinitionen

Wie sich im weiteren Verlauf der Arbeit – insbesondere bei der semantischen Analyse – zeigte, reicht die VB6-Syntax nicht aus, um alle für die Analyse relevanten Informationen über die Schnittstellen zu repräsentieren. Daher wurde das Format um zusätzliche Schlüsselwörter erweitert, die besondere Eigenschaften repräsentieren können. Beispielsweise ist es möglich, dass COM-Komponenten ausgewählte Schnittstellen in Visual Basic 6 global zur Verfügung stellen. Ihre Methoden und Propertys können dann ohne irgendeine weitere Qualifizierung überall im VB6-Quelltext angesprochen werden. Dieses Verhalten wird durch das *AppObject*-Attribut, das es in der Visual Basic 6 Syntax nicht gibt, gesteuert. Auf dem gleichen Weg wird übermittelt, ob es sich bei einer Komponente um eine *Control* – ein Steuerelement der Benutzeroberfläche – handelt. Ist dies der Fall, erweitert Visual Basic 6 die Schnittstelle intern um weitere Standardfunktionen, indem es die Schnittstelle um die Propertys und Methoden der `VBControlExtender`-Klasse erweitert. Diese und weitere Informationen sind für die korrekte Namensauflösung in der semantischen Analyse von Bedeutung und müssen daher auch im Zwischenformat, das zur Übertragung der Daten dient, repräsentiert werden.

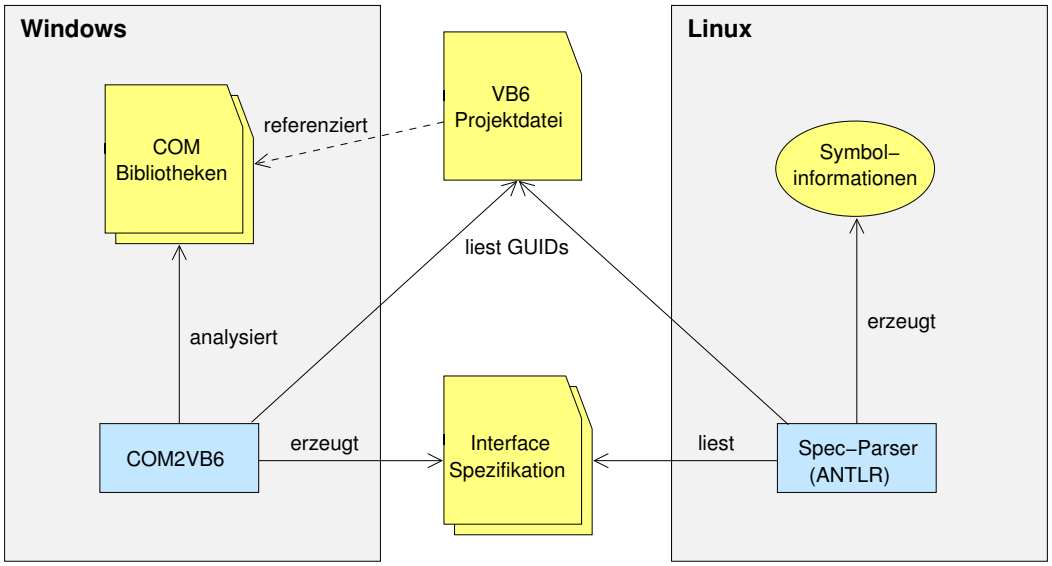

Abbildung 4.6: Daten und Aufgaben der VB6-Analyse

Visual Basic nimmt teilweise Manipulationen an den Typbibliotheken vor, wie soeben am Beispiel der Control beschrieben. Die veränderten Typelibs werden in einer Datei mit der Endung „oca“ abgelegt. Gibt es eine solche Datei, ist das tatsächlich im Quelltext verwendete Interface nur dort entnehmbar. Visual Basic ändert Teilweise die Namen von Bibliotheken und speichert diese in den OCA-Dateien. Daher versucht das Analysewerkzeug entsprechende OCA-Dateien zu finden, analysiert diese ebenfalls, erstellt eine Spezifikationsdatei und referenziert diese anhand ihrer GUID in der Spezifikation der eigentlichen COM-Komponente, die von der Projektdatei referenziert wird.

Das Format besteht daher aus einem Prolog, der den Namen der Bibliothek, die GUID und gegebenenfalls die OCA-GUID enthält. Darauf folgen die Schnittstellen und Typen der Bibliothek, die auch jeweils mit ihrer GUID angegeben sind, da diese unter Umständen in der semantischen Analyse benötigt wird. Das Format ist dabei einfach gehalten und soll hier nur anhand Listing 4.13 exemplarisch dargestellt werden. Eine vollständige Grammatikdefinition findet sich im Anhang dieser Arbeit sowie in Form der digitalen Abgabe, wo der Parser als *VB6LibSpecParser* bezeichnet wird.

```
Library VJSharpExtensibilityLib
GUID {D3809EA4-7877-4D5F-B50A-6DA64AC9882E}
Class IJSharpEventsRoot {E6FDF86F-F3D1-11D4-8576-0002A516ECE8} : dual, dispatchable
    Property VJSharpBuildManagerEvents (Optional ByVal pProject As Object) As Object
    Property VJSharpProjectItemsEvents (Optional ByVal pFilter As Object) As Object
    Property VJSharpProjectsEvents As Object
    Property VJSharpReferencesEvents (Optional ByVal pProject As Object) As Object
End Class
```

Listing 4.13: Beispiel für eine COM-Schnittstellenspezifikation (gekürzt)

Listing 4.13 Zeigt ein Interface, das nur aus einer Klasse besteht. Bei den langen Zahlenreihen handelt es sich um GUIDs. Die Klassendefinition endet mit den Attributen der Klasse, die mit einem Doppelpunkt abgetrennt sind.

Die Tatsache, dass ein menschenlesbares Format gewählt wurde, hat einen weiteren Grund. Auf diese Weise ist es ohne Weiteres möglich derartige Definitionen von Hand zu definieren. Dies wird bei der Analyse beispielsweise genutzt, um primitive Typen oder andere Dinge zu definieren, die in Visual Basic zwar vorhanden sind, jedoch in keiner Typbibliothek be-

schrieben werden. Das Spezifikationsformat bietet also zusätzlich eine Möglichkeit, Typen und Funktionen der Analyse bekanntzumachen.

4.3 Semantische Analyse

Die bisher beschriebenen Aufgabenbereiche der VB6-Analyse dienten dem Zweck, die benötigten Daten aus den Quelltexten und Typbibliotheken auszulesen und in ein strukturiertes Format zu bringen, das eine systematische Analyse erlaubt. Diese hat nun die Aufgabe, die für den RFG relevanten Informationen daraus abzuleiten und für die Generierung des Graphen bereitzustellen. Aus den Anforderungen, die ich in den vorigen Kapiteln erarbeitet habe, geht hervor, dass hierfür alle deklarierten Symbole und ihre Beziehungen ermittelt werden müssen. Ebenso gilt es, die Metriken über die analysierten Programme zu erheben.

Bevor die semantischen Aspekte von VB6-Programmen untersucht werden können, muss zunächst jedoch das benötigte Wissen über die Semantik der Sprache vervollständigt werden. Die verfügbare Dokumentation liefert nur sehr allgemeine Informationen über viele Aspekte, die für diese Untersuchung von Bedeutung sind. Es ist vor allen Dingen unklar, wie Visual Basic die Namen von Bezeichnern auflöst. Die semantische Analyse muss dieses Verhalten jedoch nachbilden. Daher leite ich aus den bekannten Informationen, eigenen Annahmen und Testfällen zunächst ein Modell für die Namensregeln her, das als Grundlage für die Nachbildung der Namensauflösung dienen kann. Hierbei verwende ich viele Einzelinformationen, die zwar an sich keine neuen Erkenntnisse darstellen, in so konzentrierter und vollständiger Form wie im hier vorgestellten Modell jedoch nicht aufzufinden waren.

Danach werde ich die Datenstrukturen beschreiben, die zur Verwendung kommen, um die benötigten Informationen festzuhalten und weitere Analysen darüber zu betreiben. Neben der Symboltabelle, die die Informationen über alle im Projekt vorkommenden Symbole aufnimmt, erläutere ich hierbei auch einige Besonderheiten des abstrakten Syntaxbaums, die für die Zwecke der semantischen Analyse hinzugefügt wurden. Auf Basis dieser Grundlagen kann der Ablauf der Analyse dann näher beschrieben werden.

4.3.1 Geltungsbereiche und Namensräume

Um alle Daten für den RFG zu erheben, muss eine Namensauflösung aller im Quelltext verwendeter Bezeichner betrieben werden. Wie schon in Abschnitt 3.3 beschrieben, lassen sich die Aufrufe und Verwendungen von Symbolen nur ermitteln, wenn alle Bezeichner ihren Symbolen zugeordnet werden. Dazu ist eine Namensauflösung erforderlich, die die Semantik von Visual Basic 6 korrekt nachbildet. Zunächst muss also klar sein, wie dies in der Sprache funktioniert.

Leider ist die Dokumentation auch hier nicht vollständig. In den Recherchen zu dieser Arbeit konnte ich nur sehr allgemeine Aussagen über die Geltungsbereiche von Symbolen finden. Sie reichen nicht aus, um diesen Vorgang korrekt nachzubilden. Aus der Dokumentation ist zu entnehmen, dass es grundsätzlich drei Geltungsbereiche gibt:

- Ein globaler (programmweiter) Geltungsbereich.
- Jedes Modul definiert einen eigenen Geltungsbereich.
- Jede Prozedur definiert einen eigenen Geltungsbereich.

Diese Bereiche sind als Schalenmodell zu verstehen, indem die oben gelisteten Namensräume sich von oben nach unten umschließen. Dabei kann ein verschachtelter Geltungsbereich

die Symbole eines darüberliegenden durch eigene gleichnamige Symbole verdecken. Innerhalb einer Prozedur ergeben sich keine weiteren begrenzten Geltungsbereiche für neu deklarierte Bezeichner. Selbst in Schleifen oder beispielsweise `If`-Blöcken deklarierte Symbole fallen ab ihrer Deklaration in den Geltungsbereich der gesamten Prozedur. Eine Variable, die in einer Schleife deklariert wird, ist also auch nach Beendigung aller Iterationen durch die Schleife noch immer vorhanden und verwendbar.

Ein Modell für Symbole und ihre Geltungsbereiche

Die drei oben genannten Abstufungen bilden ein unvollständiges Modell der Namensregeln in Visual Basic 6. Beispielsweise fehlt eine Möglichkeit, die Symbole aus Bibliotheken einzubringen, die komplexeren Regeln unterliegen – insbesondere in Bezug auf die Zusammensetzung des globalen Geltungsbereichs. Das tatsächliche Verhalten von Visual Basic muss daher in Form des bereits erwähnten Modells rekonstruiert werden. Als Ausgangspunkt verwendete ich dazu die allgemeinen Aussagen der Dokumentation. Aufgrund von Annahmen, die auf Erfahrungen mit anderen Programmiersprachen basieren und Beobachtungen in Visual Basic 6 habe ich ein Grundmodell erstellt, das ich ähnlich wie bei der Grammatikerstellung durch Tests auf Fehler untersucht und korrigiert habe.

Vorgehensweise bei der Modellerstellung

Das Grundmodell basierte dabei weitestgehend auf Vermutungen, da sich die semantischen Aspekte einer Sprache nur sehr begrenzt in einem automatisierten Prozess herleiten lassen. Davon ausgenommen sind die Vorgaben, welche Bezeichner sich auf Modulebene überschneiden dürfen und welche nicht. Diese Frage habe ich geklärt, indem ich zunächst alle möglichen Arten von Symbolen (Variablen, Prozeduren, Typen etc.) auf Modulebene ermittelt habe und für alle möglichen Paarungen aus dieser Menge automatisch Testfälle mit gleichen Bezeichnern generiert habe. So ergaben sich 126 Testprojekte, die probeweise mit dem VB6-Compiler übersetzt wurden und dabei im Falle eines Namenskonfliktes entsprechende Fehlermeldungen erzeugten. Das Programm zur Testgenerierung, die generierten Tests sowie das Testprotokoll sind in der digitalen Abgabe enthalten.

Die testbasierte Verfeinerung des Grundmodells wurde vorgenommen, indem eine dem Modell entsprechende Namensauflösung implementiert und mit den Beispielprogrammen erprobt wurde. Generell konnten hierbei zwei Arten von Fehlern auftreten:

1. Bezeichner werden als falsche Symbole aufgelöst.
2. Bezeichner werden gar nicht aufgelöst.

Fall zwei ist relativ einfach zu erkennen, weil er zu einem Fehler im Auflösungsprozess führt. Der erste Fall ist dagegen schwieriger zu identifizieren, da er keinen Laufzeitfehler hervorruft. Solche Fehler sind nur dann recht einfach wahrnehmbar, wenn sie beim Auflösen eines Bezeichners inmitten einer Qualifikation auftreten und im weiteren Verlauf der Auflösung zu Folgefehlern des zweiten Typs bei später folgenden Qualifikationselementen führen. Daher müssen die Ergebnisse der Analyse auch manuell mit den Quelltexten verglichen werden. Hierzu bieten der RFG und Gravis eine sinnvolle Präsentationsform, weshalb schon früh eine einfache RFG-Generierung geschaffen wurde, um Analyseergebnisse in ihrer RFG-Form betrachten zu können.

Die Herleitung der Semantik von Visual Basic 6 beruht insgesamt also ebenfalls nicht auf exakten Verfahren, sondern stellt eine Annäherung dar. Zudem können hierbei nur solche semantischen Aspekte aufgefunden werden, die in den Beispielprogrammen auch tatsächlich enthalten sind.

Das erarbeitete Modell

Das Modell, das ich auf diese Weise erstellt habe, basiert auf vier ineinander verschachtelten Geltungbereichen. Neben den schon aus der Dokumentation hervorgehenden Prozedur- und Modulbereichen wird, wie in Abbildung 4.7 zu sehen, je ein Bereich für das Projekt beziehungsweise eine Bibliothek und für das gesamte Programm hinzugefügt.

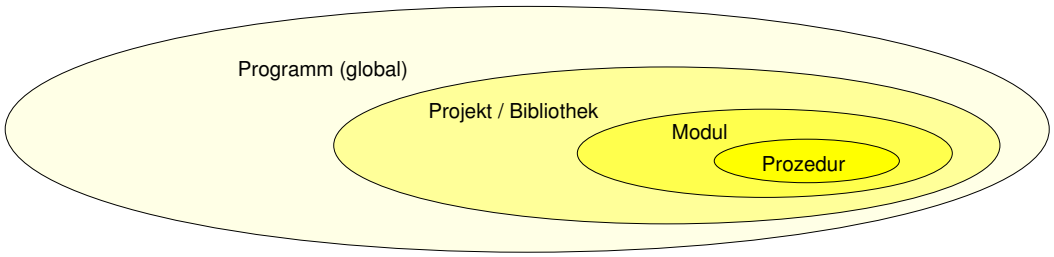

Abbildung 4.7: Schalenmodell der Geltungsbereiche in VB6

Eine wesentliche Neuerung ist hierbei die Einführung von *Projekt* und *Bibliothek* als gleichwertige Konzepte. Unter Projekt verstehe ich den Quelltext, also den tatsächlich eigenen Programmanteil. Mit Bibliothek ist dagegen eine COM-Komponente gemeint, die in das Programm eingebunden wird. Ihre Gleichwertigkeit verdeutlicht sich dadurch, dass ein VB6-Projekt in eine COM-Komponente übersetzt werden kann. Faktisch enthält also auch die Bibliothek die gleichen Geltungsbereiche, jedoch mit der Einschränkung, dass der Prozedurbereich nicht einsehbar ist. In der Semantik sind beide Konzepte daher eng verwandt und bilden Geltungsbereiche, die nebeneinander im Programm bestehen. Beide verfügen jeweils über eigene globale Symbole, die sich nach bestimmten Regeln zu einem quasi superglobalen Programmbereich vereinen. Dies ist der *Programm*-Bereich.

Überschreitung der Bereichsgrenzen

Symbole, die in einem Geltungsbereich definiert wurden, können die Bereichsgrenzen sowohl nach außen als auch nach innen durchbrechen und somit auch Gültigkeit in anderen Bereichen erhalten. Von außen nach innen geschieht dies automatisch. Ein Symbol gilt auch in allen Bereichen, die der eigene Bereich umschließt. Diese Vererbung von Symbolen im Inneren findet jedoch nur statt, sofern der innere Bereich nicht selbst ein gleichnamiges Symbol deklariert, das das globalere überschreibt. Im Prozedurbereich gelten Symbole aus höheren Geltungsbereichen solange bis dort die Deklaration eines gleichnamigen Symbols erfolgt.

Den umgekehrten Weg vollziehen globale Symbole, die auf Modul- und Projektebene (hier Modulnamen) deklariert werden können. Sie erweitern ihre Gültigkeit in die äußeren Bereiche, solange es keine anderen gleichnamigen globale Symbole anderen Ursprungs gibt. Kollidieren zwei gleichnamige Symbole in einem Geltungsbereich, so hängt die Konsequenz davon ab, wo dies geschieht. Innerhalb von Projekten eliminieren sich die globalen Symbole gegenseitig, so dass sie ihre globale Natur verlieren und nur noch mittels einer vollständigen Qualifizierung über ihren Modulnamen ansprechbar sind.

Außerhalb des Projekts gilt das First-Come-First-Served-Prinzip. Hier haben die Symbole des Quelltextes Priorität, danach folgen die Bibliotheken in der Reihenfolge, in der sie in das Projekt eingebunden wurden (und in der sie in der Projektdatei gelistet sind). Das Symbol mit der höheren Priorität wird übernommen, während das mit der niedrigeren nur noch durch eine vollständige Qualifizierung anhand des Modul- und gegebenenfalls Bibliotheksnamens referenzierbar ist. Der als *Programm* bezeichnete Geltungsbereich ist also der tatsächlich globale Geltungsbereich über das Programm.

Diese Überschreitungsregeln sind so zu verstehen, dass zuerst globale Symbole nach außen propagiert werden und erst dann alle Symbole von außen nach innen.

Eindeutigkeit von Namen

Die Identität eines Symbols hängt in Visual Basic allein von seinem Bezeichner ab. Gleichnamige Prozeduren mit unterschiedlicher Signatur können beispielsweise nicht deklariert werden. Allerdings wird diese Eindeutigkeit nicht zwischen allen Symbolen gefordert. Durch den zuvor beschriebenen automatisierten Test habe ich vier getrennte Gruppen von Bezeichnern ermitteln können, innerhalb derer Namen eindeutig sein müssen. Dies sind die Projekt- und Modulnamen, die Namen von Typen, die Namen von Events und schließlich die Namen aller weiteren auf der Modulebene deklarierbaren Symbole. Letzteres sind nach den RFG-Bezeichnungen Variablen, Konstanten, Member und Methoden, die ich zur einfacheren Bezeichnung fortan als *Membersymbole* bezeichnen werde, da sie immer Mitglied eines Moduls, Typs oder einer Prozedur sein müssen. Diese Bezeichnung wird auch in der Implementierung einheitlich verwendet.

Diese Gruppierungen haben bei der Überdeckung von Symbolen aus anderen Geltungsbereichen jedoch nur begrenzte Bedeutung. So lässt sich auf Prozedurebene jeder Modul- und Projektname durch Bezeichner, die dort oder im umgebenden Modul deklariert werden, überdecken. Die Konsequenz eines solchen Falls ist, dass das überdeckte Modul oder Projekt nicht mehr anhand seines Namens referenziert werden kann. Events und Typen sind dagegen nicht durch Symbole eines anderen Typs überdeckbar, da ihre Bezeichner in den Quelltexten stets nur an für sie reservierten Stellen auftreten können. In der Symboltabelle werden Membersymbole, Typen und Events daher in getrennten Listen geführt. Modul- und Prozedurnamen kommt eine später erläuterte Sonderbehandlung zu.

Dabei ist anzumerken, dass es einige wenige Sonderfälle zu geben scheint, die dem im Allgemeinen erkannten Muster widersprechen. Ein Beispiel hierfür sind Aufzählungstypen (Schlüsselwort `Enum`), die in Visual Basic 6 immer einen Namen tragen, der allerdings nicht zur Qualifizierung der enthaltenen Aufzählungskonstanten verwendet werden kann. Der Name eines Enums ist ausschließlich der Name eines Typs und darf nicht in Ausdrücken verwendet werden. Zumindest ist dies in der Regel der Fall. In dem Modul, in dem die Aufzählung deklariert wurde, allerdings nicht. Dort überdeckt der Name der Aufzählung alle anderen Bezeichner völlig. Da jedoch kein Compilerfehler auftritt, wenn man Symbole deklariert, die wegen dieser Eigenart eigentlich gar nicht verwendbar sind, gehe ich davon aus, dass es sich hierbei eher um einen Bug in der Sprache handelt, als um ein beabsichtigtes Verhalten. Sofern solche Ausnahmen bekannt waren, habe ich sie bei der Namensauflösung berücksichtigt. Ich stelle sie aus Gründen der Übersichtlichkeit hier jedoch nicht im Einzelnen als Teil des Modells vor.

Standardmodul- und Bibliotheksnamen

Die Namen von Standardmodulen und Bibliotheken sind ein Sonderfall dieses Modells, der näher betrachtet werden muss. Beide Namen dienen ausschließlich zur Qualifizierung anderer Bezeichner und fallen aus dem allgemeinen Schema heraus. Auf Modulebene kollidieren sie zwar mit dort definierten, gleichnamigen Symbolen und werden dabei überdeckt, andererseits überdecken sie dafür gleichnamige globale Membersymbole im Projekt- und Programmbereich. Dies lässt vermuten, dass Visual Basic diese Namen getrennt von den Membersymbolen verwaltet, die getrennte Namenslisten je nach Geltungsbereich in unterschiedlicher Reihenfolge prüft und dem jeweils zuerst gefundenen Symbol den Vorrang gibt.

4.3.2 Datenstrukturen

Nachdem ein Regelwerk für Bezeichner feststeht, kann eine Datenstruktur in Form einer Symboltabelle definiert werden, die konkrete Ausprägungen dieses Modells repräsentieren kann. In diesem Abschnitt werde ich zudem einige Modifikationen an den von ANTLR generierten Syntaxbäumen vorstellen, die zur Unterstützung der semantischen Analyse dienen. Die Symboltablle und die abstrakten Syntaxbäume bilden durch eine vielfältige gegenseitige Verlinkung in Form von Zeigern und zum Teil sogar durch die Objektidentität eine eng miteinander verknüpfte Struktur.

Symboltabelle

Die Symboltabelle wird durch mehrere Klassen, die sich gegenseitig referenzieren, modelliert. Sie ist also keine Tabelle im eigentlichen Sinn. Ich verwende den Begriff dennoch, da er sinnbildlich für das Konzept einer systematischen Symbolverwaltung steht (vgl. [Dud01, S. 649]). Kern des Modells sind die Klassen `Symbol` und `Scope`, deren Instanzen einzelne Symbole beziehungsweise Geltungsbereiche repräsentieren. `Symbol` ist darauf ausgelegt, jedes Symbol darstellen zu können und verfügt zu diesem Zweck über alle Attribute, die die verschiedenen Arten von Symbolen besitzen können. Der Typ des Symbols wird durch das Attribut `kind` definiert. Somit werden nahezu alle Symbole eines Programms durch Instanzen dieser Klasse abgebildet, Module ebenso wie lokale Variablen. Ich habe die Symbole nicht als Hierarchie von Klassen modelliert, da sie sich nicht sinnvoll nach benötigten Attributen gruppieren ließen. Das Resultat wäre entweder auf eine Klasse pro Symbolart oder eine sehr komplexe Vererbungshierarchie hinausgelaufen. Beide Fälle hätten eine Unmenge von Cast-Operationen oder gar Mehrfachvererbung in der Implementierung erfordert.

Allen Symbolen ist – unabhängig ihres Typs – gemein, dass sie das `name`-Attribute zum speichern ihres Bezeichners verwenden. Viele der weiteren Attribute sind spezifisch für besondere Symboltypen. Während ein Modul im Wesentlichen nur über einen Namen und einen selbst definierten Geltungsbereich in Form eines referenzierten `Scope`-Objektes verfügt, werden für Prozeduren Informationen wie die Sichtbarkeit, die Quelltextposition, der Rückgabetyp, und eine Liste der Parameter (die selbst wiederum `Symbol`-Objekte sind) gespeichert. Eine vollständige und dokumentierte Auflistung aller Attribute kann der mit dem Tool *Doxygen*[7] generierten Code-Dokumentation auf der beiliegenden CD entnommen werden. Sie würde an dieser Stelle den Rahmen sprengen.

Symbole wie Prozeduren, Module oder Typen definieren selbst neue Geltungsbereiche. Diese werden durch `Scope`-Objekte dargestellt, die Maps über alle Symbole des Geltungsbereichs

[7] Doxygen: `http://www.stack.nl/~dimitri/doxygen/`

beinhalten. Als Schlüssel dienen dabei die Bezeichner. Die Elemente der Listen sind Zeiger auf die jeweiligen `Symbol`-Objekte. Für die zuvor identifizierten Namensbereiche für Typen, Events und Membersymbole werden hier drei getrennte Listen verwaltet. Zu jeder Liste bietet der `Scope` Methoden zum Hinzufügen neuer Symbole und zur Abfrage, ob ein bestimmtes Symbol enthalten ist. Zudem können Kopien der vollständigen Listen abgefragt werden.

Die `Symbol`- und `Scope`-Objekte bilden eine verkettete Struktur, die einen Baum darstellt (sofern man den globalen `Scope` außenvor lässt). Generell wird dabei jedes Symbol in dem `Scope` aufgeführt, in dem es deklariert wurde. Um einen (nicht qualifizierten) Namen, der in einem bestimmten Geltungsbereich verwendet wird, aufzulösen, muss daher nur vom aktuellen `Scope` ausgehend der Baum der `Scopes` nach oben durchlaufen werden, bis ein `Scope` gefunden wird, der das gesuchte Symbol enthält. Abbildung 4.8 zeigt eine vereinfachte Darstellung der Modellierung, Abbildung 4.9 ein exemplarisches Objektdiagramm dazu. Zur besseren Übersichtlichkeit stelle ich in den UML-Diagrammen keine Attribute und Operationen dar.

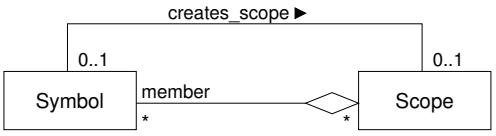

Abbildung 4.8: Klassenmodell der Symboltabelle (stark vereinfacht zur Vermittlung des allgemeinen Konzeptes)

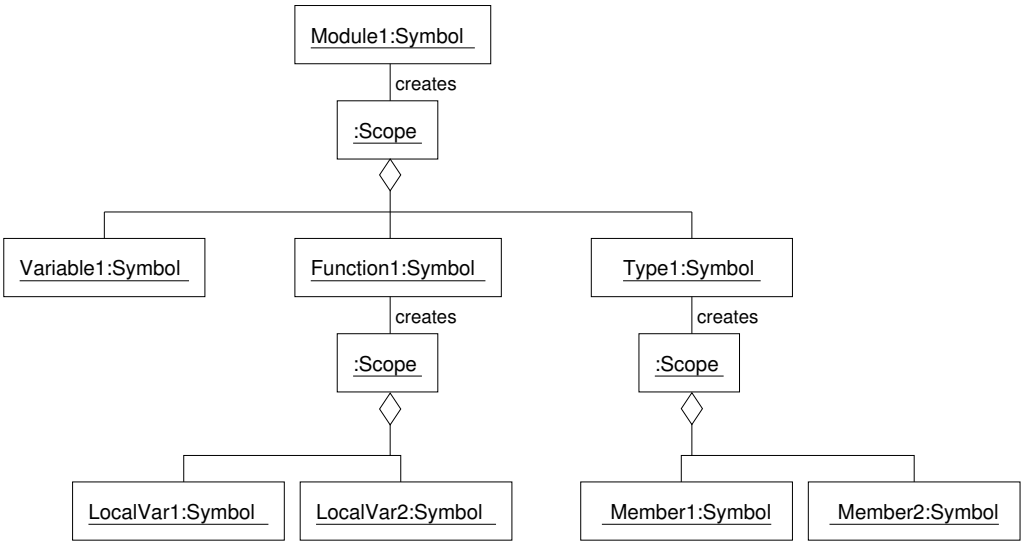

Abbildung 4.9: Beispielhaftes Objektdiagramm einer Symboltabelle (vereinfacht)

In einigen Geltungsbereichen gibt es besondere Regeln, wie bereits am Beispiel der `Enum`s in Modulen beschrieben. Diese Abweichungen vom Standardverhalten werden durch Spezialisierungen der `Scope`-Klasse, die ein entsprechendes Verhalten in den Methoden zum Einfügen neuer Symbole implementieren, modelliert. Die Namen von `Enums` werden auf Modulebene beispielsweise mit absoluter Priorität in die Liste der Membersymbole aufgenommen, um das außergewöhnliche Verhalten nachzubilden.

Standardmodule, Klassen und Formulare stellen zwar ebenfalls Symbole dar und werden auch als `Symbol`-Objekte dargestellt, allerdings sind sie unterschiedlich zu behandeln. Klassen und Formulare definieren *Typen*, die Teil des globalen Namensraumes sind, während Standardmodule ausschließlich einen Geltungsbereich definieren. Ihr Name kann ausschließlich zur Qua-

lifizierung anderer Elemente verwendet werden. Er bezeichnet keinen Typ. Insgesamt werden die folgenden Informationen über Module benötigt:

- Ihre `Symbol`- und `Scope`-Objekte.
- Eine Liste aller Standardmodule für die Namensauflösung.
- Eine einheitliche Struktur zum Iterieren über alle Module ungeachtet ihres Typs. (Zum Durchlaufen der gesamten Symboltabelle.)

Diese Anforderungen werden von der Klasse `ProgramUnit` erfüllt, die sowohl das Projekt, als auch Bibliotheken repräsentieren kann. Sie verfügt über eine Liste, die Zeiger auf alle Module des Quelltextes beziehungsweise der Bibliothek beinhaltet und somit einen Eintrittspunkt für ein systematisches Durchlaufen der gesamten Symboltabelle bietet. Dieses kann an der `ProgramUnit` beginnen und von dort aus dem Baum folgen, den die `Symbol`- und `Scope`-Objekte aufspannen. So ergibt sich eine Baumstruktur, die es ermöglicht, über die gesamte Symboltabelle einer `ProgramUnit` zu iterieren und dabei jedes Symbol und jeden Scope genau einmal zu besuchen.

Für die Zwecke der Namensauflösung wird eine weitere Liste geführt, die ausschließlich die Standardmodule enthält. Schließlich hält die `ProgramUnit` einen globalen Namensraum über das Projekt beziehungsweise die Bibliothek, der unter anderem die `Symbol`-Objekte für Klassen und Formulare in seiner Typauflistung führt. Somit sind die Symbole für Module Teil mehrerer Strukturen, die unterschiedlichen Zwecken dienen. Abbildung 4.10 erweitert das vorige Beispiel um die `ProgramUnit` und den globalen Geltungsbereich.

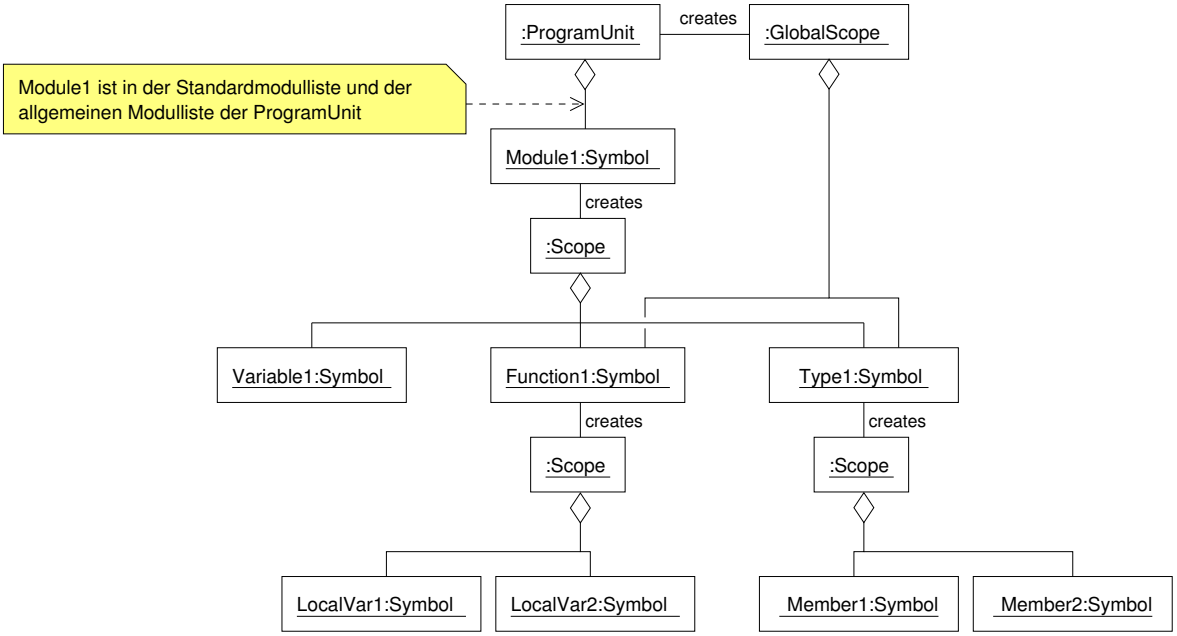

Abbildung 4.10: Beispielhaftes Objektdiagramm einer Symboltabelle um die Projektebene erweitert (vereinfacht)

Sowohl das Projekt als auch die Bibliotheken verfügen selbst über Namen. Auch sie sind Symbole, die jedoch nicht durch `Symbol`-Objekte abgebildet werden. Um `Symbol`- und `ProgramUnit`-Objekte in der Namensauflösung einheitlich verwenden zu können, erben beide von der Klasse `NamedSymbol`. Diese definiert mit dem Namen das grundlegende Attribut für Symbole und erlaubt die erwünschte einheitliche Handhabe der beiden anderen Klassen.

Es fehlt noch die Modellierung der Ebene des gesamten Programms. Diese wird durch das `SymbolTable`-Objekt abgedeckt, das die `ProgramUnit` für die Symbole des Quelltextes referenziert und über eine geordnete Liste aller Bibliotheken, ebenfalls in Form von `ProgramUnit`-Objekten, verfügt. Einen Geltungsbereich, der den Programm-Bereich des Modells abbildet, hat sie jedoch nicht. Der Grund hierfür ist die Tatsache, dass die Symbole aus dem Quelltext im Programm-Geltungsbereich Vorrang haben. Die globalen Bibliothekssymbole werden nur dann aufgenommen, wenn sie nicht mit denen des Projekts in Konflikt stehen.

Daher kann der Programm-Geltungsbereich als Erweiterung des Projekt-Bereichs verstanden werden. Um dies zu modellieren, verfügt die Klasse für den globalen Scope (`GlobalScope`) über zusätzliche Methoden, die es erlauben, die Symbole von Bibliotheken nach den besonderen Verdrängungsregeln, die ich im Modell beschrieben habe, in den Projekt-Geltungsbereich aufzunehmen. Standardmäßig führt das Hinzufügen eines bereits im `Scope` des Projekts enthaltenen Symbols dazu, dass keines der beiden kollidierenden Symbole übernommen wird und der Name in eine Ausschlussliste aufgenommen wird. So wird das Verhalten abgebildet, das innerhalb des Projektes üblich ist. Die Funktionen zum Hinzufügen eines Bibliothekssymbols weisen dagegen bereits vorhandene Symbole einfach ab und belassen das vorhandene in der Symbolliste. Vorausgesetzt, dass die Bibliothekssymbole in der richtigen Reihenfolge hinzugefügt werden, ist somit zugleich sichergestellt, dass die Verdrängungregeln des Modells korrekt umgesetzt sind.

Die `SymbolTable`-Klasse dient damit also nur dazu die einzelnen Programmeinheiten zusammenzufassen und den Zugriff darauf zu ermöglichen. Demnach ergibt sich das in Abbildung 4.11 dargestellte Schema für die gesamte Symboltabelle. Zur besseren Übersichtlichkeit habe ich wie schon zuvor auch hier auf die Darstellung von Attributen und Operationen verzichtet. Die Darstellung bezieht sich zudem nur auf die Struktur der Symboltabelle hinsichtlich der Geltungsbereiche. Weitere Informationen, wie beispielsweise die Referenzierungen zwischen den `Symbol`-Instanzen, sind nicht dargestellt.

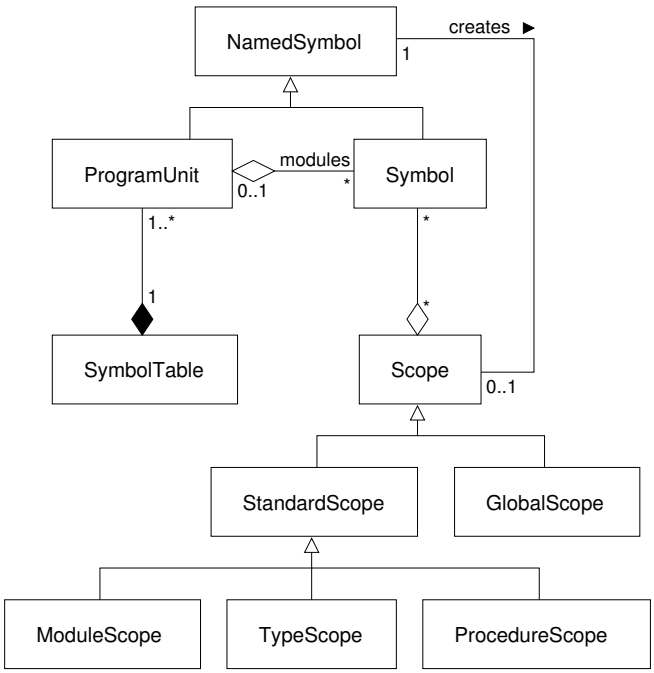

Abbildung 4.11: Klassendiagramm der Symboltabelle (Ausschnitt)

Abstrakte Syntaxbäume

Die abstrakten Syntaxbäume werden in dieser Arbeit vor allem dazu benötigt, die Bezeichner des Quelltextes zu analysieren. Dazu ist es wichtig, dass neben den Bezeichnern selbst auch die Prozedur und der Geltungsbereich, in dem sie sich befinden, aus dem AST hervorgeht. Die Symboltabelle wird nicht auf Basis der Syntaxbäume erstellt, sondern schon in den Actions des Parsers. Generell bestehen die ASTs, die ANTLR generiert, nur aus Knoten einer einzigen Klasse. Sie sind in diesem Sinne also homogen. Um die Analyse der ASTs zu vereinfachen, habe ich für die Knoten, die von besonderem Interesse sind jedoch spezielle Knoten-Klassen implementiert. Diese erleichtern es zum einen, die Knoten in einem Visitor-Pattern (vgl. [GHJV94, S. 331]) anhand ihres Typs zu erkennen und unterschiedlich zu behandeln, zum anderen ermöglichen sie es, knotenspezifische Funktionalität hinzuzufügen.

Dies dient vor allem zwei Zwecken. Die `Symbol`- und `Scope`-Objekte der Symboltabelle verfügen über Spezialisierungen, die zugleich die Funktionalität von AST-Knoten erben (deren Namen sind jeweils um das Suffix `AST` erweitert, siehe Abbildung 4.12). Dadurch wird es möglich diese anstelle der Deklarationen in den AST einzuhängen und dort auf diese Weise den Wechsel von Geltungsbereichen zu kennzeichnen. Ein `Scope`-Knoten führt dabei immer einen Teilbaum an, dessen Folgeknoten allesamt in dem durch den `Scope` repräsentierten Geltungsbereich liegen. AST und Symboltabelle bilden dabei zwei Strukturen, die über ihre eigenen Zeigerstrukturen betrachtet und durchlaufen werden können Hierbei ist es aber jederzeit möglich an den Punkten, die sowohl Teil des ASTs als auch der Symboltabelle sind, zwischen beiden Strukturen zu wechseln. Hierzu ist höchstens eine Cast-Operation notwendig.

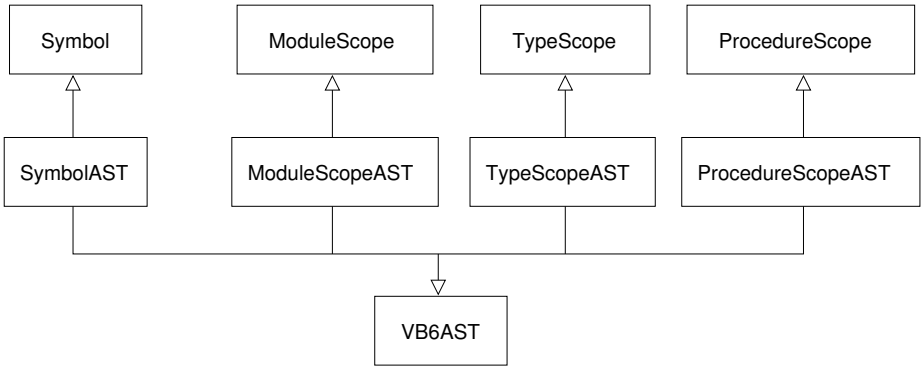

Abbildung 4.12: Vereinigung von Symbolen, Geltungsbereichen und dem AST

Hierbei sind nicht alle Symbole und Geltungsbereiche auch zwangsläufig Teile des Syntaxbaumes. Dies ist für die Symbole aus Bibliotheken der Fall. Der COM-Parser erstellt keine Syntaxbäume, sondern direkt eine Symboltabelle. Daher sind AST und Symboltabelle als getrennte Strukturen mit Überschneidungspunkten zu verstehen.

Neben der Symboltabelle und dem AST gibt es zwei weitere Klassen, die verwendet werden, um Bezeichner und Qualifikationen, die aus Bezeichnern bestehen, zu modellieren (dabei ist anzumerken, dass Qualifikationen auch aus nur einem Bezeichner bestehen dürfen). Die Namen der Klassen sind entsprechend `Identifier` und `QualifiedId`. Sie dienen dazu, die Untersuchung von Qualifikationen in der Namensauflösung zu vereinfachen. Dort wird es notwendig sein, über Qualifikationen zu iterieren und ihre Bezeichner schrittweise den entsprechenden Symbolen zuzuordnen. Problematisch ist dabei, dass Bezeichner und Qualifikationen in unterschiedlichen Darstellungsformen auftreten. Zum einen gibt es sie in Form von Syntaxbäumen, die zusätzliche Token, wie die geklammerten Argumentenlisten, enthalten. In der COM-Analyse stellen die Typen, die in den Deklarationen enthalten sind, ebenfalls Qualifika-

tionen dar, die jedoch nicht als Bäume dargestellt werden. Stattessen werden die `Identifier`- und `QualifiedId`-Objekte hier direkt im Parser erstellt.

`Identifier` und `QualifiedId` bieten daher ein einheitliches Interface für den Umgang mit Bezeichnern und Qualifikationen und abstrahieren von der eigentlichen Darstellungsform. Für den AST gibt es Spezialisierungen namens `IdentifierAST` und `QualifiedIdAST`, die die Baumstruktur einer Qualifikation als Datengrundlage verwenden (siehe Abbildung 4.13). Sie liefern zudem bereits Informationen, die ein umständliches Vor- und Rücklaufen über den Baum ersparen. So gibt die `Identifier`-Klasse Auskunft über den Operator, der zur Qualifizierung verwendet wurde und aus `QualifiedId` ist die Art der Qualifizierung erkennbar. Diese zeigt an, ob es sich um einen Typnamen, ein Event oder ein Membersymbol handelt.

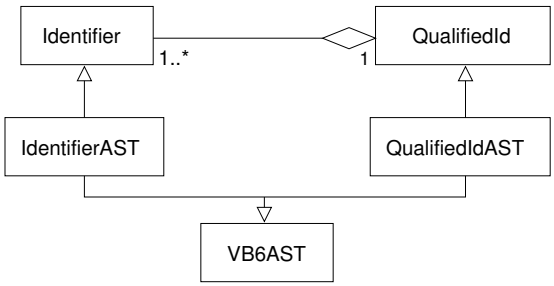

Abbildung 4.13: Klassendiagramm der Bezeichner- und Qualifikationsklassen (Ausschnitt)

In dieser Form erlaubt es der AST zur Auflösung von Bezeichnern mit einem Visitor über den AST zu laufen, den aktuellen Geltungsbereich in einem Stapel mitzuführen und die für den RFG relevanten Qualifikationen zu erkennen. Hier sind Typangaben, die beispielsweise nach einem `New`-Schlüsselwort auftreten, nicht von Bedeutung – Events und Membersymbole hingegen schon. Die Klassifizierung erlaubt es, jede Qualifizierung korrekt zu behandeln, ohne den Kontext anhand der Baumstruktur während der Namensauflösung nochmals untersuchen zu müssen.

Die Ergebnisse der Namensauflösung werden in den `Identifier`-Knoten in Form von Zeigern auf die jeweiligen Symbole, die sie darstellen, gespeichert. Damit ergeben AST und Symboltabelle nach der Analyse eine stark verkettete gemeinsame Struktur, die alle für den RFG benötigten Informationen bereithält und nur noch ausgelesen werden muss. Abbildung 4.14 verdeutlicht die Zusammenhänge von Quelltext, AST und Symboltabelle an einem Beispiel. Gleichbenannte Objekte haben jeweils die gleiche Identität.

Weitere Spezialisierungen der Standardklasse für AST-Knoten dienen dazu, Programmelemente, die für die Namensauflösung und die Gewinnung von Informationen über die Ablaufsemantik bedeutend sind, einfach anhand eines Visitors identifizieren zu können. Sie werden beispielsweise für die Anweisungen `With`, `RaiseEvent` und `ReDim` eingesetzt und werden später noch genauer erläutert.

4.3.3 Analysevorgang

Die Analyse auf Basis der beschriebenen Datenstrukturen erfolgt stets über ein vollständiges VB6-Projekt. Die Sprache kennt keinen Mechanismus, der sicherstellt, dass im Quelltext verwendete Symbole zuvor deklariert wurden. Es ist möglich, dass in einem Modul unqualifizierte Symbole verwendet werden, die erst in einem später analysierten Modul als global deklariert werden. Zudem ist es möglich, Variablen implizit durch deren Verwendung zu deklarieren. Es müssen alle globalen Symbole bekannt sein, um in jedem Fall entscheiden zu können, ob ein

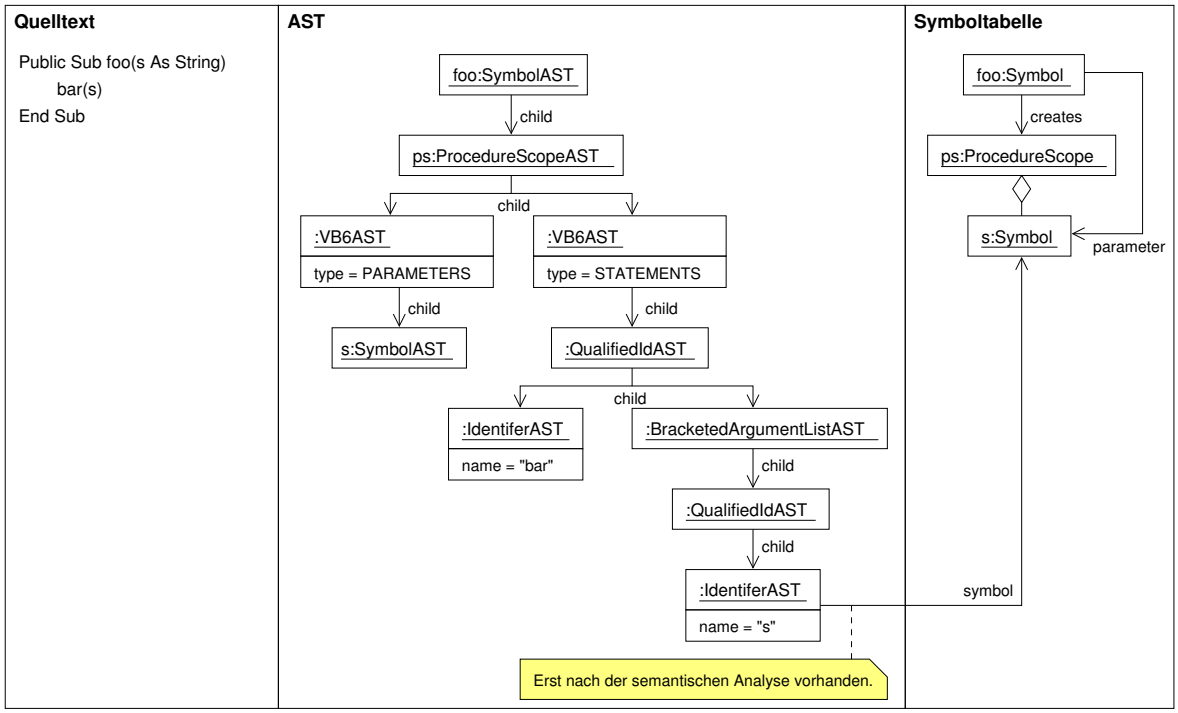

Abbildung 4.14: Eine Prozedur in Quelltext-, AST- und Symboltabellen-Repräsentation

Symbol implizit deklariert oder ein globales referenziert wird.

Aus diesen Gründen kann die Identität von Symbolen nur dann eindeutig ermittelt werden, wenn zuvor eine vollständige Symboltabelle erstellt wurde. Die semantische Analyse von VB6-Programmen kann also nicht in nur einem Durchlauf über die Quelltexte erfolgen. Sie teilt sich in die folgenden Schritte auf:

1. **Sammeln der Symbolinformationen:** In diesem ersten Schritt werden die Deklarationen der Symbole im Quelltext und den COM-Spezifikationen analysiert und die entsprechenden `Symbol`- und `Scope`-Objekte erstellt. Aus Effizienzgründen erfolgt dies bereits in den Actions der jeweiligen Parser. Auch die Erhebung von Metriken wird in diesem Zuge durchgeführt.

2. **Auflösen von Typen in der Symboltabelle:** Die Typen von Symbolen können nicht schon während der Konstruktion der Symboltabelle in ihre Symbole aufgelöst werden, da die erste Referenzierung eines Typs bereits vor seiner Deklaration erfolgen kann. Da die Information, von welchem Typ ein Symbol ist, jedoch bei der folgenden Namensauflösung benötigt wird, muss dieser Schritt zwischengeschoben werden.

3. **Auflösen von Bezeichnern im Quelltext:** Nachdem die Symboltabelle vollständig aufgebaut ist, kann mit dem Auflösen der Bezeichner im Quelltext fortgefahren werden, indem ein Visitor den AST durchläuft und die dortigen `QualifiedId`s auf der Prozedurebene auflöst.

Im Folgenden werde ich zunächst die einzelnen Schritte der Namensauflösung näher beschreiben. Die Erhebung der Metriken behandle ich im Anschluss daran.

4.3.3.1 Sammeln der Symbolinformationen

Die Parser erzeugen die `Symbol`- und die dazugehörigen `Scope`-Objekte immer dann, wenn sie die Deklaration eines Symbols analysieren. Das `SymbolTable`-Objekt wird zu Beginn des Analysevorgangs eines Projektes erstellt. Mit ihm zusammen entsteht zugleich die `ProgramUnit` für die Symbole des Quelltextes. Die `ProgramUnit`s für die Bibliotheken werden für jede vom Projekt eingebundene COM-Komponente erzeugt.

Die Erzeugung der Objekte für die Symbole einer `ProgramUnit` erfolgt während des Parsens der einzelnen Module. Einhergehend mit dem Top-Down-Vorgehen des Parsers wird die Symboltabelle dabei von oben nach unten konstruiert. Die Hierarchie der Geltungsbereiche kann problemlos mit der fortschreitenden Linksableitung erstellt werden, da sichergestellt ist, dass die höher gelegenen Geltungsbereiche vor den verschachtelten erzeugt werden.

In die Symboltabelle werden neben der Art des Symbols (bspw. Variable oder Function) auch die aus seiner Deklaration hervorgehenden Attribute im `Symbol`-Objekt aufgenommen. Dies können Eigenschaften wie die Sichtbarkeit, Parameter oder die Anwesenheit von Modifikatoren wie `Static` sein. Die Typen werden vorerst nur in Form von `QualifiedId`-Objekten festgehalten. Im Falle der Quelltextanalyse also als Zeiger auf den `QualifiedIdAST`-Knoten im AST.

Im Ergebnis liegen nach den Quelltext- und COM-Analysen also eine Symboltabelle mit allen explizit deklarierten Symbolen und je ein AST für jedes Quelltextmodul vor. Um den globalen Namensraum zu vervollständigen, werden die globalen Symbole der Bibliotheken nach dem zuvor beschriebenen Muster in den globalen Namensraum des Projekts übernommen. Erst dann sind die Geltungsbereiche korrekt nachgebildet.

4.3.3.2 Auflösen von Typen in der Symboltabelle

Das Auflösen der Typen in der Symboltablle ist eine relativ einfache Aufgabe, da deren Qualifizierungen recht einfach strukturiert sind – sie bestehen ausschließlich aus Bezeichnern und Punkten und kennen keine Abkürzungsformen. Besonders unkompliziert stellt sich dies bei Typen dar, die innerhalb einer Bibliothek verwendet werden. Diese können nämlich selbst nur Typen sein, die entweder in der Bibliothek selbst oder in anderen COM-Komponenten definiert sind. In den COM-Komponenten liegen Typen zudem generell im obersten Geltungsbereich, selbst dann, wenn die Komponente in Visual Basic erstellt wurde, das die Deklaration von Typen nur innerhalb von Modulen erlaubt. Eine Typangabe kann daher entweder aus einem einzelnen Bezeichner bestehen, dann muss der Typ aus der Bibliothek selbst stammen oder ein primitiver Typ sein. Sie kann kann aber auch aus zwei Bezeichnern bestehen, wobei der erste eine Bibliothek bezeichnet und der zweite den Typ. In diesem Fall kann der Typ direkt durch seine vollständige Qualifizierung gefunden werden.

Ein wenig komplizierter ist der Vorgang bei der Auflösung von Typangaben, die aus dem Quelltext eines Programms stammen. Hier können Typen in Modulen deklariert werden, wobei der Modulname auch zur Qualifizierung verwendet werden kann. Es ist zudem möglich, Typen in einem Modul als privat zu deklarieren, während auf globaler Ebene ein gleichnamiger Typen existiert, der innerhalb des Moduls durch den lokalen Typ verdeckt wird. Die Auflösung einer Typangabe, die nur aus einem Bezeichner besteht, muss also erfolgen, indem der Scope-Baum hinaufgelaufen wird, bis zum ersten Mal ein gleichnamiger Typ aufgefunden wird.

Bei einer Typangabe, die aus zwei Bezeichnern besteht, kann der erste Bezeichner sowohl eine Bibliothek (bzw. das Projekt), als auch ein Modul bezeichnen. Tests in Visual Basic 6 ergeben, dass hierbei zunächst geprüft wird, ob eine Bibliothek vorhanden ist, die den angegebenen

Namen trägt. Erst danach wird nach einem Modul geprüft. An dieser Stelle zeigt sich erneut die Heterogenität, die in den Regeln zur Namensauflösung von Visual Basic herrscht.

Ein Typ, der mit drei Bezeichnern angegeben wird, kann nur über den Projektnamen und einen Modulnamen qualifiziert sein. Um unerwarteten Sonderfällen vorzubeugen, wird der erste Name jedoch auch zusätzlich als Bibliotheksname gesucht. Sobald ein Typ erkannt wurde, wird ein Zeiger ausgehend von dem Symbol, das den Typen referenziert, zum `Symbol`-Objekt des Typs angelegt.

Eine weitere Aufgabe, die im Zuge der Typauflösung erledigt wird, ist das Erkennen von eventbehandelnden `Sub`-Prozeduren. Diese sich durch einen Namen gekennzeichnet, der sich aus dem Namen einer Modulvariable und dem Name des zu behandelnden Events zusammensetzt. Da dies der einzige Hinweis auf eine Eventbehandlung ist, muss jeder `Sub`-Prozedurname auf dieses Schema geprüft werden. Behandelt eine Prozedur ein Event, so wird ihrem `Symbol`-Objekt ein Zeiger auf das `Symbol` des Events hinzugefügt. Aus dieser Information können später die `handled_by`-Kanten generiert werden. Hierzu muss lediglich die Richtung des Zeigers invertiert werden.

4.3.3.3 Auflösen von Bezeichnern im Quelltext

Das Auflösen der Bezeichner im Quelltext wird in einem Visitor realisiert, der in einer Tiefensuche über die ASTs aller Modulquelltexte iteriert. Der Parser hat den AST bereits in einer Form vorbereitet, die es einfach macht, die aufzulösenden Qualifikationen zu erkennen und den zum Typ der Qualifikation passenden Algorithmus zu wählen. Dieser ist für Events und die sonstigen Symbole wie Variablen und Prozeduren unterschiedlich. Den Typ erkennt der Parser bereits am Kontext, in dem die Qualifikation aufgefunden wurde und hält ihn im `QualifiedId`-Objekt fest.

Der Visitor führt den aktuell gültigen Geltungsbereich in einem Stapel mit, um den Ausgangspunkt für die Auflösung von Namen zu kennen. Neben den allgemeinen Geltungsbereichen protokolliert er zudem die `With`-Statements und ihre Verschachtelungen. Wie ich bereits zuvor in diesem Kapitel erläutert habe, ermöglichen es `With`-Statements, die Mitglieder eines komplexen Typs in einer Kurzform zu referenzieren, indem die Qualifizierung mit einem unären Qualifizierungsoperator begonnen wird. Für die Auflösungen solcher Kurzformen gelten spezielle Geltungsbereiche, nämlich der Scope des Typs, der Gegenstand des `With` ist. In Listing 4.14 muss die Auflösung von `.Name` mit dem `Scope` des Person-Typs begonnen werden, die Auflösung von `p2.Name` dagegen beginnt von dem aktuellen Geltungsbereich aus. Daher werden die Geltungsbereiche der `With`-Statements ebenfalls in einem Stapel mitgeführt. Jedes öffnende `With`-Statement führt zum Hinzufügen eines Scopes, der bei Verlassen des `With`-Blocks wieder vom Stapel angenommen wird. Wann welcher `Scope` zur Auflösung herangezogen wird, geht aus dem Beginn der Qualifizierung hervor.

```
Dim p1 As New Person
Dim p2 As New Person

With p1
    .Name = "Jan"
    p2.Name = "Jan"
End With
```

Listing 4.14: Kontextabhängige Reservierung von Schlüsselwörtern

Das Auflösen von Eventnamen ist eine sehr einfache Aufgabe. Die einzige Einsatzmöglichkeit des Namens besteht – neben der bereits behandelten Kennzeichnung von eventbehandelnden `Sub`-Prozeduren – darin, Events mit dem `RaiseEvent`-Statement auszulösen. Dies kann nur

in der Klasse geschehen, in der das Event deklariert wird. Daher bleibt nur der umgebende Modulscope, um das Eventsymbol zu suchen.

Die Auflösung von Qualifikationen, die Variablen und Prozeduren enthalten, ist dagegen der schwierigste Teil der gesamten semantischen Analyse. Hier können alle Sonderformen und Abkürzungen auftreten, die es allesamt zu behandeln gilt. Ich werde zunächst den Aufbau und die möglichen Bestandteile einer solchen Qualifikation beschreiben und dann den Algorithmus zu deren Auflösung vorstellen.

Generell ist eine Qualifikation in einzelne Bezeichner unterteilt, denen eine beliebige Zahl von geklammerten Argumentenlisten folgen kann. Diese einzelnen Bestandteile werden durch die binären Qualifikationsoperatoren („. “ und „! “) getrennt. Selbige Operatoren können in ihrer unären Form auch am Beginn einer Qualifikation in einem `With`-Statement stehen. Soweit die syntaktische Struktur.

Die Semantik der Bezeichner hat Einfluss auf den Geltungsbereich, in dem nach dem Symbol des jeweils nächsten Bezeichners zu suchen ist. Der aktuelle Geltungsbereich, in dem der Bezeichner aufgefunden wurde, dient nur dann als Ausgangspunkt für die Namensauflösung, wenn der erste Bezeichner eine Variable oder Prozedur ist. Wird eine absolute Qualifizierung vorgenommen, indem Bibliotheks- oder Modulnamen vorangestellt werden, so geben diese den Geltungsbereich zur Suche des Symbols vor. Im weiteren Verlauf der Qualifizierung wird der Geltungsbereich, in dem das folgende Symbol zu suchen ist, durch den (Rückgabe)-Typ des vorhergehenden Qualifikationselements definiert.

Die Auflösung von Bezeichnern allein ist daher das geringere Problem. Interessant sind dagegen die schon in Abschnitt 2.1.3 erwähnten geklammerten Argumentenlisten. Aus der Syntax geht nicht immer eindeutig hervor, ob es sich bei einer solchen Klammerung um eine Liste von Funktionsparametern, den Zugriff auf ein Arrayfeld, den Zugriff auf eine Standardeigenschaft, ein Dictionary Lookup Operator oder ein geklammertes erstes Argument einer `Sub`-Prozedur handelt. Alle diese Fälle werden durch den gleichen Knoten im AST (`BRACKETED_ARGUMENT_LIST`) dargestellt. Zur besseren Übersicht stellt Listing 4.15 die semantische Zusammensetzung von Qualifikationen in einer EBNF nach [ISO96] dar. Um diese Darstellung in einer EBNF zu ermöglichen, verwende ich die nicht näher definierten Syntaxregeln `identifier`, `libname`, `modulename`, `procedure arg list`, `array access` und `dictionary lookup`, die Bestandteile der Qualifikation mit bestimmter Semantik bezeichnen. Die letzten drei sind hierbei die unterschiedlichen Formen von geklammerten Argumentenlisten.

```
qualification     = [ [ prefix ], operator], element, { operator, element }
operator          = "." | "!"
prefix            = libname, ".", modulename | libname | modulename
element           = identifier, [ bracketed suffix ]
bracketed suffix  = procedure arg list, { array access | dictionary lookup }
```

Listing 4.15: Zusammensetzung von Qualifikationen in EBNF-Notation

Das Ziel der Auflösung ist es, allen Bezeichnern einen Zeiger auf ihr Symbol in der Symboltabelle hinzuzufügen. Hierbei gibt es zwei Sonderfälle zu berücksichtigen: Die Dictionary Lookups stellen Funktionsaufrufe dar, die später auch im RFG abzubilden sind. Daher kann der AST-Knoten für eine geklammerte Argumentenliste ebenfalls ein Symbol referenzieren und somit angeben, welche Funktion durch ihn aufgerufen wird. Im Falle der !-Notation (bspw. `fruits!banana` statt `fruits.Item("banana")`) wird der Bezeichner, der den Schlüssel des abgerufenen Elements benennt, verwendet, um auf die Zugriffsmethode zu verweisen. Abbildung 4.15 zeigt ein Beispiel für diese Verweisstrukturen, wobei die gestrichelten Linien die Zeiger darstellen, die vom AST zur Symboltabelle angelegt werden würden.

Die Auflösung solcher Qualifizierungen erfolgt, indem über alle Elemente iteriert und dabei

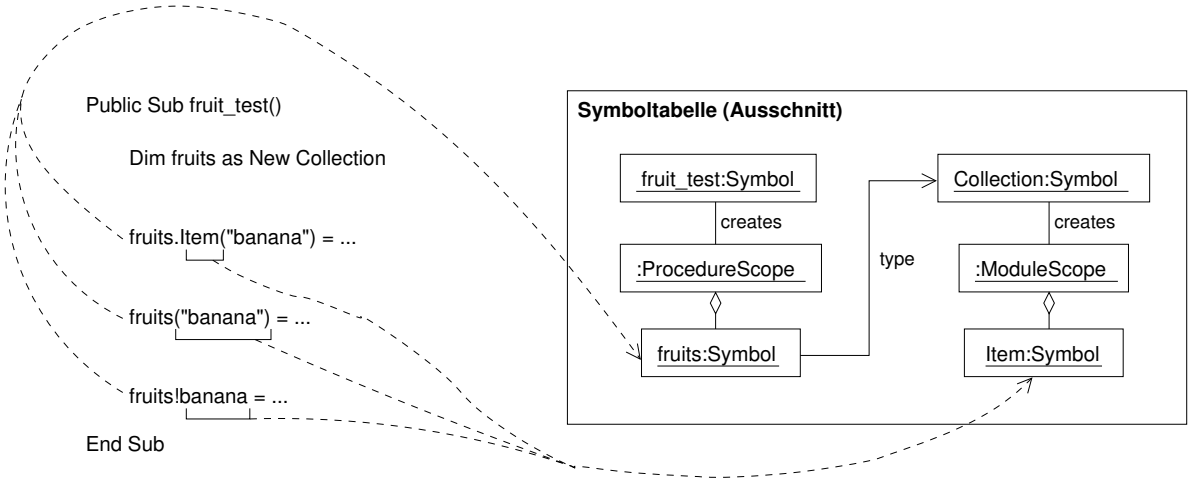

Abbildung 4.15: Abbildung von Qualifizierungselementen auf die Symboltabelle

der Geltungsbereich mitverfolgt wird. Dieser wechselt mit jedem Bezeichner, aber auch mit den Klammerungen, die einen Arrayzugriff oder einen Dictionary Lookup darstellen. Dies ist der Fall, da diese wiederum einen bestimmten Typen zurückliefern. Für jedes Element wird zunächst der Bezeichner aufgelöst und dann alle folgenden Klammerungen. Da die lokalen Bezeichner Modul- und Bibliotheksnamen verdecken können, wird zuerst versucht, ein `element` im Sinne der obigen EBNF zu erkennen. Scheitert dies, kann mit der Suche nach Modul- und Bibliotheksnamen fortgefahren werden, sofern diese den Regeln nach noch auftreten können. Somit setzt sich ein Algorithmus zusammen, der in Listing 4.16 in Pseudocode beschrieben wird. Die Grundidee ist es hierbei, in jeder Iteration alle validen Möglichkeiten zur Auflösung eines Elements zu durchlaufen und bei der erfolgreichen Erkennung zum nächsten Iterationsschritt zu springen. Läuft eine Iteration bis zum Ende der Schleife, so wurde das Element auf keinem Weg erfolgreich aufgelöst.

```
General rules:
    - Keep track of the current scope while resolving the ids in the qualification
    - Keep track of a current ProgramUnit to where module names will be looked up
    - If an id was resolved break out of the current iteration an continue with next
      id
    - If resolving fails for an id, stop resolving the qualification
    - 'bracketed argument list' is abbreviated as 'bal'

Preconditions:
    - expect library names
    - expect module names
    - current scope is the scope the qualified id was found in or
      the current with scope if there is one
    - current ProgramUnit is the projects ProgramUnit


for each id in qualification do:

    if there is no current scope:
        abort with error
    end if

    if id is first of qualification and not in a with statement:
        try to find the symbol by walking up the scopes, beginning with the current
          one
    else:
        try to find the symbol in the current scope only
    end if

    if id was found as a member symbol:
```

```
        add a pointer from the id to the symbol
        get the scope of the symbols type as the new current scope
        don't expect module or library names anymore

        if id is followed by bal in the AST:
            obtain a pointer on the first bal following the id
            if the id was some sort of procedure and the paramter count fits:
                set the pointer to the next bal
            end if

            while AST pointer points to a bal do:
                if the symbol creating the current scope is an array:
                    move AST pointer to next bal before next qualification element
                    continue while
                end if

                try to find a default or dictionary lookup method in current scope
                if found such a method symbol:
                    add a pointer from the bal to the symbol
                    get the scope of the symbols type as the new current scope
                    move AST pointer to next bal before next qualification element
                end if
            end while

        end if

        continue with next id
    else:
        if expecting module names:
            if expecting library names:
                try to find the id as lib name in SymbolTable
                if found id as lib name:
                    add a pointer from the id to the ProgramUnit
                    save the library as current ProgramUnit
                    get the global scope from the library as new current scope
                    don't expect library names anymore
                    continue with next id
                end if
            else:
                try to find the id as module name in the current ProgramUnit
                if found id as module symbol:
                    add a pointer from the id to the symbol
                    get the modules scope as the new current scope
                    don't expect module names anymore
                    continue with next id
                end if
            end if
        end if
    end if

    //if this point is reached resolving the id failed completely
    abort with error

end for each
```

Listing 4.16: Kontextabhängige Reservierung von Schlüsselwörtern

Der Sonderfall `ReDim`

Die `ReDim`-Anweisung dient in Visual Basic dazu, die Dimensionen eines existierenden Arrays nachträglich zu ändern. Demnach wäre ihre Semantik für den RFG nicht von Bedeutung, da Arraydimensionen dort nicht abgebildet werden. Allerdings ist es möglich – obgleich Microsoft in der Dokumentation dringend davon abrät – diese Anweisung zu verwenden, um ein gänzlich neues Array zu deklarieren. Problematisch hierbei ist, dass mit `ReDim` die Dimensionen von globalen Variablen, die in völlig anderen Modulen deklariert werden, veränderbar sind. Hier besteht also die gleiche Problematik wie bei den implizit deklarierten Variablen –

`ReDim`-Deklarationen sind erst eindeutig erkennbar, wenn eine vollständige Symboltabelle vorliegt. Erst dann kann entschieden werden, ob eine globale Variable aus einem anderen Modul referenziert oder eine neue deklariert wird.

Der Parser lässt diese Frage daher offen und erzeugt lediglich einen speziellen Knoten für das `ReDim`-Statement. Die Auflösung von `ReDim`-Anweisungen erfolgt erst im Visitor, der die Bezeichner des AST auflöst. Dies ist möglich, da dieser zum einen über eine vollständige Symboltabelle verfügt und zum anderen, weil die durch `ReDim` deklarierten Arrays prinzipiell nur lokale Symbole einer Prozedur sein können, da `ReDim` nur in Prozeduren verwendet werden kann. Das bedeutet, dass sie bei der Auflösung von Namen, die vor ihrer Deklaration stehen, nicht beachtet werden müssen. Der Visitor, der eine Tiefensuche auf dem AST durchführt, findet ReDim-Deklarationen immer bevor die dadurch deklarierten Symbole verwendet werden können. Im Falle einer `ReDim`-Deklaration fügt der Visitor ein entsprechendes Symbol in die Symboltabelle ein.

4.3.4 Metriken

Die Bauhaus-Werkzeuge erheben Metriken nach [Axi06] auf Routinen bezogen. Die Berechnung der vorgegebenen Metriken *LOC*, McCabe und Halstead ist relativ einfach zu realisieren, da hierzu lediglich verschiedene Programmartefakte abgezählt werden müssen. Für *LOC* und McCabe muss daher nur sichergestellt sein, dass diese zum Zeitpunkt der Metrikerhebung auch vollständig vorhanden sind. Die Halstead-Berechnung erfordert zudem, dass die Unterscheidung zwischen Bezeichnern und Schlüsselwörtern bereits erfolgt ist und dass alle Wörter und Zeichen der Sprache zur Verfügung stehen, um die Zahl der Operatoren ermitteln zu können.

All diese Bedingungen lassen sich ohne größere Probleme schon im Parser bereitstellen und die Zählung kann in Form von Actions realisiert werden. Eine Lösung schon im Parser birgt den Vorteil, dass nicht alle für die Metriken benötigten Token auch in die ASTs übernommen werden müssen. Somit lässt sich ein unnötiges Aufblähen der Bäume und damit ein unnötig großer Speicherverbrauch vermeiden.

Wie bereits erwähnt lassen sich die Metriken, beziehungsweise deren Ausgangsdaten, durch einfaches Abzählen realisieren. Der Parser verfügt zu diesem Zweck über Zählervariablen, die die aktuellen Werte speichern (bei Halstead ist dieses Verfahren etwas komplexer, siehe unten). Da die Zählung prozedurbasiert erfolgt, werden alle Zähler neu initialisiert, sobald eine Prozedurdeklaration erreicht wird. Am Ende einer Prozedur werden die erhobenen Daten aus den Zählern ausgelesen und an das `Symbol`-Objekt der Prozedur in der Symboltabelle annotiert. Dies hat zu diesem Zweck Felder, die diese Daten aufnehmen können. Im Folgenden beschreibe ich das Vorgehen zur Erhebung der jeweiligen Metriken.

4.3.4.1 Lines of Code

In Abschnitt 2.4.1 habe ich bereits beschrieben, welche Maße in Bezug auf die Programmzeilen im Bauhaus-Projekt verwendet werden. Die Erhebung der reinen *LOC* ist dabei außerordentlich einfach. Da die Token Informationen über ihre Codeposition beinhalten, kann schlicht die Nummer der Anfangszeile - 1 von der Nummer der Endzeile subtrahiert werden, um den gewünschten Wert zu erhalten.

Die Frage, ob eine Zeile ausführbaren Programmcode enthält, also die Berechnung von *Code*, erfolgt durch die Verwendung eines booleschen Wertes und der Betrachtung der Zeilenenden, die im Tokenstrom enthalten sind. Der boolesche Wert zeigt an, ob seit dem letzten Zeilenum-

bruch die Produktion `statement`, die alle möglichen Anweisungen kapselt, aufgerufen wurde. Ist dies der Fall, so lag eine als *Code* zu zählende Zeile vor. Diese Entscheidung wird beim Erkennen des Zeilenumbruchs in einer Action getroffen.

Die Zählung der Kommentare erfolgt auf ähnlich Weise. Sie macht sich die Tatsache zunutze, dass Kommentare in Visual Basic grundsätzlich nur am Ende einer Zeile stehen dürfen. Es gibt keine begrenzten Kommentare, hinter denen weiterer ausführbarer Code folgen kann. Daher lässt sich die Frage, ob eine Zeile einen Kommentar enthält, klären, indem an jedem Token, der einen Zeilenumbruch Repräsentiert, geprüft wird, ob vor ihm ein Kommentar im Tokenstrom lag. Wie ich bereits bei der Quelltextanalyse beschrieben habe, sind die Kommentare im Tokenstrom versteckt, wobei jeder Token seine vorhergehenden und folgenden versteckten Token referenziert. Diese Funktionalität stellt ANTLR bereits zur Verfügung.

Somit ist allerdings erst geprüft, ob die Zeile einen Kommentar enthält, jedoch nicht, ob sie nicht auch Anweisungen enthält und somit nicht zu *Only_Comment* zu zählen ist. Letzteres lässt sich allerdings einfach nachvollziehen, indem auf den für die Berechnung von *Code* eingeführten booleschen Wert zugegriffen wird, der besagt, ob die Zeile ausführbaren Code enthielt. Die Abzählung der leeren Zeilen (*Empty*) ist nach der Erhebung der anderen Werte nicht mehr notwendig, da sie sich aus $LOC - Code - Only\_Comment$ berechnen lässt.

4.3.4.2 McCabe

Die zyklomatische Komplexität kann wie in Abschnitt 2.4.3 beschrieben durch die Menge ihrer binären Entscheidungspunkte plus 1 berechnet werden. Daher muss festgelegt werden wann in einem VB6-Programm solche Situationen vorliegen. Hierfür habe ich die folgenden Schlüsselwörter gewählt:

- Schleifen:
  - `For`
  - `For Each`
  - `While`
  - `Do`
  - `Loop`
- Bedingte Entscheidungspunkte:
  - `If`
  - `ElseIf`
  - jedes `Case` in einem `Select Case`-Block
- Bedingte Sprünge:
  - `On` *Bedingung* `Goto`

Die Schleifen prüfen jeweils eine Bedingung, die entweder zu einer Iteration durch die Schleife oder einer Programmfortführung nach der Schleife führen. Das heißt sie stellen eine binäre Verzweigungsmöglichkeit dar. Das `If`-Statement ist ein einfacher Verzweigungspunkt. Hierbei sind mit `Else` eingeleitete Alternativen nicht als Entscheidungspunkt zu sehen, da die Entscheidung allein in der Prüfung der `If`-Bedingung erfolgt. Anders ist es mit dem `ElseIf`. Hier wird eine Entscheidung getroffen, wenn der Kontrollfluss diesen Punkt erreicht. Letztere

Frage ist allerdings für die McCabe-Berechnung nicht von Bedeutung, da schlicht die Anzahl der Verzweigungspunkte des Kontrollflusses gezählt werden.

`Select Case` lenkt den Kontrollfluss anhand eines Wertes und gibt mit `Case` Verzweigungspunkte für explizite Werte an, sowie einen optionalen Standardfall. Das `Select Case`-Schlüsselwort hat selbst keine verzweigende Eigenschaft, jedes `Case` kann dagegen direkt mit einem `If` verglichen werden und muss gezählt werden. Die Standardeigenschaft korrespondiert dagegen mit dem `Else`-Schlüsselwort uns ist daher nicht mitzuzählen. Hierbei ist anzumerken, dass Visual Basic 6 das `Select Case`-Statement immer nach der Ausführung eines Falls verlässt. Hierzu ist keine `break`-Anweisung wie in anderen Sprachen vorhanden. Das Behandeln mehrerer Fälle mit dem gleichen Programmcode wird in VB6 durch die Angabe mehrerer Bedingungen für ein `Case` realisiert. Diese Bedingungen werden implizit verodert. Daher stellt jedes `Case` einen abgeschlossenen Pfad im Kontrollflussgraphen dar und kann direkt gezählt werden.

Auch Sprünge können mit `On` *Bedingung* `Goto` *Sprungziel* bedingt ausgeführt werden. Ist die Bedingung erfüllt, wird der Sprung an das (früh gebundene) Sprungziel ausgeführt, ansonsten nicht. Neben dieser Anweisung bestehen zudem die ähnlich erscheinenden `On Error Goto` *Sprungziel* und `On Error Resume (Next)`. Diese Anweisungen haben jedoch keinen statisch nachvollziehbaren Einfluss auf die Programmausführung, sondern legen das Programmverhalten im Falle eines Fehlers fest. So kann eine im Fehlerfall anzuspringende Sprungmarke angegeben werden oder das Programm dazu veranlasst werden, mit der gleichen beziehungsweise der nächsten Anweisung fortzufahren. Ihr Effekt ist laufzeitgebunden und nicht sinnvoll in einem Kontrollflussgraphen darstellbar, da nahezu jede Anweisung potentiell einen Fehler verursachen könnte.

`On` *Bedingung* `Goto` verursacht dagegen an der Stelle, in der es im Programm steht, tatsächlich einen Sprung, sofern die Bedingung erfüllt ist. Da das Ziel des Sprungs zum Übersetzungszeitpunkt feststeht und nicht berechnet sein darf, liegt hier eine eindeutig nachvollziehbare Kante für den Kontrollflussgraphen vor. Somit kann das Auftreten dieser Anweisung als binäre Verzweigung verstanden werden und ist daher zu zählen.

Wie in [WM96, S. 25] beschrieben sind auch Logikoperatoren mitzuzählen, wenn deren Auswertung so optimiert ist, dass der rechte Ausdruck nur dann ausgewertet wird, wenn der linke nicht schon den Ausgang festgelegt hat. Sprich im Fall von `False And eineFunktion()` würde die Funktion nicht mehr aufgerufen, da die Verundung schon durch das `False` bereits nicht mehr zu Wahr auswerten kann. Die Operatoren in VB6 haben keine solche Semantik, wie durch einen Test, der dem soeben genannten Beispiel folgt, sehr einfach nachvollzogen werden kann - `eineFunktion()` wird in jedem Fall ausgeführt. Daher liegt hier keine Verzweigung im Kontrollfluss vor und somit ist die zyklomatische Zahl nicht zu erhöhen.

4.3.4.3 Halstead

Zur Berechnung der Halstead-Metriken muss zunächst festgelegt werden, welche Token als Operanden zu verstehen sind und welche als Operatoren. Im Bauhaus-Projekt werden alle Bezeichner und konstanten Werte als Operanden verstanden, alle weiteren Token als Operatoren. In dieser Arbeit verfahre ich daher auf die gleiche Weise. Die zentralen Werte der Halstead-Metrik sind die in Abschnitt 2.4.2 Variablen N_1, N_2, μ_1 und μ_2, die Anzahl unterschiedlicher Operanden und Operatoren und ihre jeweilige Gesamtzahl darstellen. Sie reichen aus, um alle Berechnungen durchzuführen.

Die Berechnung dieser Werte kann erst im Parser erfolgen. Abschnitt 4.1.4.1 beschreibt die Produktion, in der die Unterscheidung zwischen Bezeichnern und Schlüsselwörtern erfolgt. Vor diesem Punkt in der Analyse können die Halstead-Werte nicht berechnet werden. Eine

weitere Anforderung für ihre Berechnung ist aber auch, dass *alle* Token, die der Lexer erzeugt (mit Ausnamhe der Leerzeichen), betrachtet werden. Beide Anforderungen lassen sich erfüllen, indem die `consume`-Methode des ANTLR-Parsers überschrieben wird. Diese wird immer dann aufgerufen, wenn ein Token erkannt wurde und aus dem Tokenstrom verworfen wird. Hier kann jedes Token betrachtet.

Die einzelnen Token und ihre Lexeme nimmt der Parser in zwei Abbildungen auf – je eine für Operatoren und Operanden. Darin wird jedes Lexem, also die Zeichenkette des Tokens, einer Zahl zugeordnet, die angibt wie oft es im Tokenstrom enthalten war. Die Anzahl unterschiedlicher Operatoren beziehungsweise Operanden entspricht daher der Anzahl an Abbildungen, die Gesamtzahl erhält man durch Addieren aller Zahlen.

Die Werte für N_1, N_2, μ_1 und μ_2 werden am `Symbol`-Objekt für die Prozedur gespeichert. Die Berechnung der einzelnen Halstead-Werte, wie beispielsweise das Volumen geschieht dynamisch aufgrund der im `Symbol`-Objekt gespeicherten Halstead-Grundwerte. Die Formeln dazu habe ich in Abschnitt 2.4.2 bereits beschrieben.

4.3.4.4 Zusätzliche Metriken

In [Axi06] werden neben den Metriken, die in der Aufgabenstellung dieser Arbeit gefordert werden, zusätzlich die maximale Verschachtelungstiefe, die Anzahl von Parametern einer Routine und die Anzahl von Aufrufen einer Routine erwähnt. Die ersten dieser drei sind ausgesprochen einfach zu erheben und werden daher ebenfalls berechnet.

Die Parameteranzahl kann dabei direkt der Symboltabelle entnommen werden. Die Verschachtelungstiefe wird, wie die anderen Metriken auch, bereits im Parser erhoben. Hierzu werden zwei Integer-Variablen verwendet, wobei eine die aktuelle Verschachtelungstiefe repräsentiert und die andere das erreichte Maximum. Als Verschachtelung wird dabei jede Kontrollstruktur angesehen, die über eine einleitende und eine abschließende Anweisung verfügt und somit einen Block definiert. Eingeschlossen sind auch solche Kontrollstrukturen, die in einzeiliger Form auftreten können, wie beispielsweise eine `If`-Anweisung mit nur einem folgenden Statement. In diesem Fall würde die Verschachtelung inkrementiert und nach der folgenden Anweisung sogleich wieder verringert werden. Zu Beginn des Anweisungsbereichs einer Prozedur werden beide Zählervariablen auf Null gesetzt. Am Ende der Prozedur wird das Maximum ausgelesen und an den Symboltabellen-Eintrag der Prozedur annotiert.

Die Anzahl der Aufrufe wird in der vorliegenden Implementierung nicht berechnet. Sie müsste an anderer Stelle als die anderen Metriken erhoben werden, da die Aufrufsemantik erst nach dem vollständigen Auflösen aller Bezeichner bekannt ist. Ein nachträglicher Einbau einer solchen Berechnung sollte im Visitor erfolgen, der die Bezeichner im AST auflöst. Die Anzahl der Aufrufe könnte in einem weiteren numerischen Attribut an der Klasse `Symbol` erfolgen, das bei jeder Referenzierung der Funktion durch einen Bezeichner (Zuweisungen von Rückgabewerten ausgenommen) inkrementiert wird.

KAPITEL 5

Generierung des RFG

Die Generierung des RFG ist von der Analyse der VB6-Programme losgelöst und basiert lediglich auf den Daten, die diese hervorbringt. Dabei handelt es sich um die vollständige Symboltabelle für ein VB6-Projekt sowie die abstrakten Syntaxbäume für alle Modulquelltexte. Alle relevanten Informationen wie Typen, Referenzierungen oder Funktionsaufrufe sind in diesen Datenstrukturen bereits durch Zeiger repräsentiert, die lediglich ausgelesen werden müssen.

In diesem Kapitel beschreibe ich die Vorgehensweise, nach der der RFG aufgebaut wird. Der RFG selbst ist in Form einer vorhandenen Ada-Bibliothek realisiert, die bereits alle benötigten Funktionen zum Aufbau des Graphen bereitstellt. Diese müssen durch eine Schnittstelle zwischen Ada und C++ aufgerufen werden. Über diese Schnittstelle wird dann der RFG in zwei Phasen der Generierung der deklarierten Symbole und ihrer statischen Beziehungen und dem Hinzufügen der Kanten, die das Programmverhalten repräsentieren, aufgebaut. Über den fertigen Graphen gilt es dann noch die nötigen Sichten zu erzeugen und das Linken mehrerer RFGs zu einem Gesamtgraphen zu ermöglichen.

Kapitelinhalt

5.1 Ada-C++ Interfacing

Eine Schnittstelle, die es erlaubt, die RFG-Bibliothek aus C++-Programmen heraus anzusprechen wurde bereits zuvor in der Diplomarbeit von Markus Müller, die ein Bauhaus-Frontend für Cobol entwickelt, erstellt [Mü]. Zwar ist dieser Code einige Zeit nicht gepflegt worden, konnte aber wiederverwendet werden, nachdem einige Neuerungen der RFG-Bibliothek nachgepflegt wurden.

Der Ada-Anteil der Implementierung ist dabei äußerst gering. Es wird lediglich ein einzelner RFG erzeugt, der von der C++-Seite aus über einige Schnittstellenmethoden um Knoten, Kanten und Attribute erweitert werden kann. Die Schnittstellenmethoden geben dabei Zeiger auf die erzeugten Knoten an die C++-Seite zurück. Sie können verwendet werden, um weitere Operationen auf dem Knoten aufzurufen – konkret das für Hinzufügen eines Attributes oder das Erstellen einer Kante zwischen zwei Knoten.

Um die Schnittstelle an die Symboltabelle anzubinden, verfügt die RFG-Generierung über eine Registratur, die die Symbole ihren RFG-Knoten-Zeigern zuordnet. Sie liefert Auskunft

darüber, ob bereits ein RFG-Knoten zu einem Symbol existiert und gibt den Zeiger zu diesem Knoten bei Bedarf zurück.

Eine weitere Funktion, die auf der Ada-Seite realisiert wurde, ist die Generierung der Sichten, zu der ebenfalls bereits Algorithmen im Bauhaus-Projekt vorhanden sind. Die Schnittstelle wurde um Funktionen erweitert, die es erlauben, diese View-Generierung von C++ aus anzustoßen. Zudem wurden neue Schnittstellenmethoden hinzugefügt, die ermöglichen, Attribute wahlweise als Zeichenkette, booleschen oder numerischen Wert an bestehende Knoten anzuhängen. Das allgemeine Funktionsprinzip der Schnittstelle blieb dabei unverändert und kann daher [Mü, S. 47] entnommen werden. Im Hinblick auf eine einfache Wiederverwendung für andere Zwecke wurde die Schnittstelle jedoch als eigene Bibliothek realisiert, so dass sie kein direkter Teil der VB6-Analyse ist.

5.2 Vorgehen bei der RFG-Generierung

Die Generierung des RFG ist ein relativ unkompliziertes Unterfangen, da alle relevanten Daten bereits vorliegen. Es muss nur eine Reihenfolge gewählt werden, die sicherstellt, dass alle RFG-Elemente, die durch Kanten referenziert werden sollen, auch tatsächlich vorhanden sind. Zudem muss die Grapherstellung die in Abschnitt 3.3 vorgegebenen Konventionen erfüllen. In diesem Abschnitt beschreibe ich die Erstellung des gesamten Graphen, wie er in der Basis-View des RFG enthalten ist.

Generell gibt es zwei Arten von Symbolen, die es zu unterscheiden gilt. Zum einen sind dies die im Programm deklarierten Symbole, die in ihren `Symbol`-Objekten das Attribut `isDefined` tragen, sowie die Symbole, die aus externen Bibliotheken stammen. Während erstere prinzipiell in den RFG übernommen werden, sind die externen Symbole nur dann aufzunehmen, wenn diese in irgendeiner Form referenziert werden. Beispielsweise wenn eine Bibliotheksfunktion aufgerufen oder eine Klasse aus einer Bibliothek instanziiert wird.

Um die deklarierten Symbole in den RFG in Form von Basiseinheiten aufzunehmen, iteriert die RFG-Generierung über den Teil der Symboltablle, der die Symbole aus den Quelltexten repräsentiert. Hierbei wird – einer Breitensuche ähnlich – jedes Symbol besucht und die entsprechende Basiseinheit mit allen Attributen erzeugt. Das Iterieren durch die Symboltabelle erlaubt es gleichzeitig, die `Enclosing`-Kanten zu erstellen, die die hierarchische Ordnung der RFG-Basiseinheiten repräsentieren.

Zu jedem Symbol werden zudem seine ausgehenden RFG-Kanten erzeugt, sofern diese aus der Symboltabelle hervorgehen. Dies gilt insbesondere für die `Of_Type`, `Return_Type` und `Parameter_Of_Type`. Hierbei kann es passieren, dass das Ziel dieser Kanten im RFG noch gar nicht in Form einer Basiseinheit vorhanden ist. In diesem Fall wird dieser Knoten bei Bedarf erstellt, so dass er referenziert werden kann. Die Einbettung in die Symbolhierarchie, sowie das Erstellen seiner ausgehenden Kanten erfolgt jedoch erst dann, wenn dieser Knoten in der Abfolge der Interaktion an der Reihe ist. Hier wird seine Erstellung dann ausgelassen und direkt mit dem Ziehen von Kanten fortgefahren.

Genauso ist es auch möglich, dass ein externes Symbol referenziert wird. Dieses muss dann ebenfalls in Form eines Knotens in den RFG aufgenommen werden. Hierzu wird die Symboltabelle nach oben durchlaufen und die Symbole, die eine `Enclosing`-Kante zu dem neu zu erzeugenden Knoten haben, miterzeugt.

Nach diesem ersten Schritt enthält der RFG alle deklarierten Symbole mitsamt ihrer Attribute sowie die von ihnen ausgehenden Kanten. Es fehlen noch die Kanten, die das Verhalten des Programms definieren, also die Aufrufe, Variablen- und Memberverwendungen sowie die

Eventsemantik. Diese Informationen werden dem AST entnommen, der zu diesem Zweck von einem Visitor durchlaufen wird.

Während die Art des Bezeichners (Member, Event, Typ) bereits im AST annotiert ist, muss für die Member-Bezeichner der Kontext, in dem ein Bezeichner steht, festgestellt werden, da die RFG-Kanten zwischen lesender (`Use`) und schreibender (`Set`) Verwendung differenzieren. Hierbei geht der Visitor grundsätzlich davon aus, dass ein lesender Kontext besteht. Diese Festlegung wird nur dann überschrieben, wenn ein Zuweisungs-Teilbaum betreten wird. Hier folgt immer eine Qualifizierung, deren letztes Element in einem `Set`-Kontext steht. In einer Zuweisung wird in VB6 prinzipiell immer nur einmal ein Wert zugewiesen und zwar dem Bezeichner, der links des Zuweisungsoperators steht. Danach sind alle folgenden `=`-Operatoren Gleichheitsprüfungen – Zuweisungen und Vergleiche verwenden den gleichen Operator in VB6.

Der Kontext ist auch bei der Referenzierung von Propertys von Bedeutung, da sich über ihn entscheidet, welche Zugriffsprozedur aufgerufen wird. Die Symboltabelle speichert für jede Property ein spezielles `Symbol`, dass das Property-Symbol darstellt und eine Liste aller Accessormethoden – sortiert nach deren Zugriffsmethode – hält. Der RFG-Generator kann anhand des Kontextes eine Methode wählen zu der er eine Aufrufkante zieht.

Zusätzlich verwendet er die `Scope`-Knoten des AST, um den aktuellen Geltungsbereich mitzuführen. Da alle Bezeichner mit ihren Symbolen verlinkt sind, kann so ermittelt werden, welche Bezeichner von welcher Routine oder Methode (erkennbar über den aktuellen Geltungsbereich) referenziert wird und die entsprechende Kante in den RFG übernommen werden.

Sollte dabei ein noch nicht im RFG enthaltenes externes Symbol referenziert werden, so wird dieses wie bereits zuvor beschrieben nach Bedarf erzeugt. Nachdem alle ASTs durchlaufen wurden, ist die Basis-Sicht des RFG vollständig erzeugt.

5.3 Sichten

Die RFG-Bibliothek des Bauhaus-Projektes enthält bereits Prozeduren, die die Call-, Module- und Environment-Views aus der Basis-View erzeugen. Diese werden auch zur Generierung der Sichten für den VB6-RFG verwendet. Durch die Erweiterung des RFG-Modells in Kapitel 3, ist der Algorithmus für die Call-View jedoch nicht mehr direkt anwendbar, da sie die neuen Konzepte der Events nicht berücksichtigt.

Bislang wurden hier alle für diese Sicht relevanten Knotentypen übernommen, sowie alle Kanten, die Spezialisierungen der `Call`-Kante sind. Die neu definierten `Event`-Knoten werden dadurch jedoch nicht in die Call-View übernommen, da es sie bislang nicht gab. Ich habe diesen Algorithmus daher dahingehend erweitert, dass er auch `Event`-Knoten in die Call-View aufnimmt. Für die neuen Kanten ist keine Änderung erforderlich, da sie bereits von der `Call`-Kante erben.

5.4 Linken

Die Werkzeuge, die in dieser Arbeit erstellt wurden, erzeugen grundsätzlich nur RFGs für einzelne VB6-Projekte. Zu den Bauhaus-Tools gehört das Programm *rfglink* (siehe [Axi06]), das mehrere RFGs zu einem zusammenfügen kann. Damit dies funktioniert, muss der Graph Informationen bereithalten, die es dem Linker ermöglichen, Knoten zweier RFGs, die das gleiche Symbol repräsentieren, auch als gleich zu erkennen. Alle Knoten enthalten zu diesem Zweck die beiden Attribute `Linkage.Name` und `Linkage.Is_Definition`. Ersteres beinhaltet

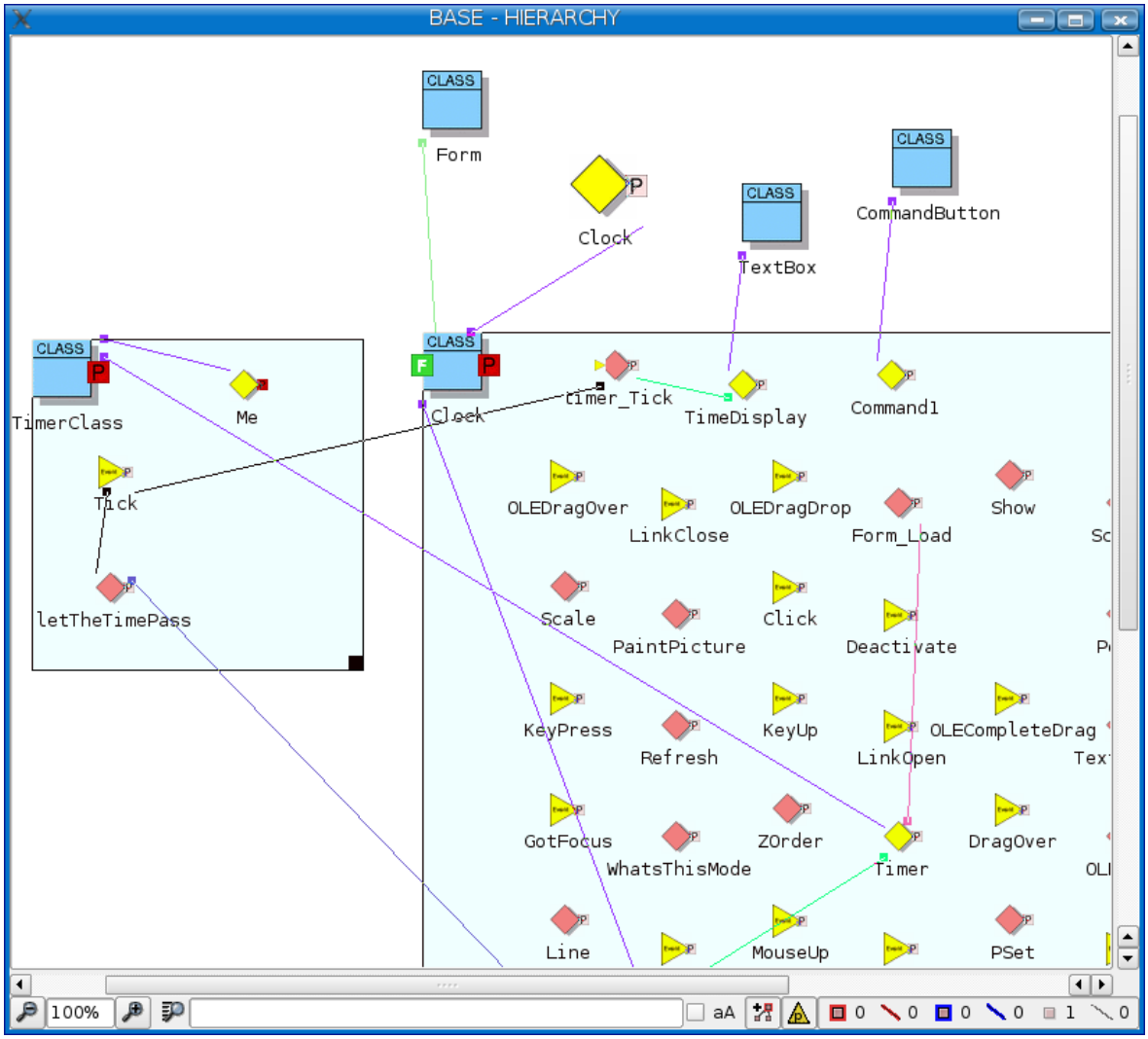

Abbildung 5.1: Der RFG eines einfachen VB6-Programms in Gravis.

einen eindeutigen Namen für einen Knoten, letzteres wird genutzt, um festzuhalten, ob ein Knoten aufgrund der Deklaration eines Symbols erstellt wurde oder nur mit dem Wissen, dass dieses Symbol im Code verwendet wird. Beim Linken werden stets die Attribute des mit `Linkage.Is_Definition` gekennzeichneten Knotens übernommen.

Dieser Fall tritt bei der Analyse von Visual Basic 6 nur bei den `Unknown_Symbol`-Knoten auf. Diese haben kein `Linkage.Is_Definition`-Attribut. Den eigentlichen Zweck kann dieses Attribut in diesem Fall jedoch nicht erfüllen, denn selbst wenn der Knoten, der die Definition darstellt, vorhanden ist, kann dieser nicht erkannt werden, da eine eindeutige Zuweisung des unbekannten Symbols zu seiner Definition generell nicht möglich ist.

Als eindeutiger Schlüssel bietet sich auf den ersten Blick die Verwendung der GUID an. Diese ist allerdings nicht immer verfügbar. Wird beispielsweise eine Bibliothek mit Visual Basic erstellt, so lässt sich ihre GUID nicht aus den Quelltexten oder der Projektdatei ermitteln oder daraus ableiten. Daher kann sie nicht als eindeutiger Bezeichner dienen, denn will man den RFG einer Bibliothek in einen anderen RFG einlinken, der diese nutzt, so müssen die Knoten für die gleichen Konzepte logischer Weise auch den gleichen `Linkage.Name` haben.

Daher wird statt der GUID der voll qualifizierte Name, bestehend aus Bibliotheksname, Mo-

dulname etc., verwendet. Dazu ist anzumerken, dass diese Namen nach den Konventionen von Visual Basic 6 und COM nicht mehrdeutig sein können. Ein Projekt kann mehrere gleichnamige Bibliotheken einbinden. Allerdings ist es eher unwahrscheinlich, dass dieser Fall eintritt. Zudem müsste nicht nur der Bibliotheksname, sondern der voll qualifizierte Name des Elements mehrfach existieren.

Im Allgemeinen kann auch der Linker direkt für VB6-RFGs übernommen werden. Jedoch muss auch hier die abgeänderte Call-View-Generierung verwendet werden, damit die Events korrekt behandelt werden.

KAPITEL 6

Ergebnisse und Zusammenfassung

In den vorhergegangenen Kapiteln habe ich Grundlagen, Anforderungen, Konzepte sowie eine konkrete Realisierung der Aufgabe beschrieben. In diesem abschließenden Kapitel sollen nun die Ergebnisse betrachtet und bewertet werden. Dies erfolgt anhand von Tests, die überprüfbare Kriterien hinsichtlich Vollständigkeit, Qualität und Robustheit der vorliegende Implementierung und der ihr zugrundeliegenden Konzepte, betrachten. Abschließend resümiere ich in einer Zusammenfassung über diese Arbeit.

Kapitelinhalt

6.1 Bewertung der Ergebnisse

Das Ergebnis dieser Arbeit ist eine Implementierung, die es ermöglicht den RFG für Visual Basic 6 Programme zu erstellen. Ich habe Methoden angewendet, die auf einer testbasierten Annhärung an die tatsächliche Syntax und Semantik der Sprache basieren. Diese sind nicht exakt und erlauben es daher nicht mit Sicherheit, zu einer vollständigen und korrekten Lösung zu kommen. Durch die vielen Besonderheiten und die sehr begrenzte Dokumentation der Sprache, ist es schwer solch einer Lösung nahe zu kommen.

In diesem Abschnitt möchte ich daher die Ergebnisse der erstellten Werkzeuge untersuchen, um festzustellen inwieweit die Annährung gelungen ist. Hierzu betrachte ich die beiden Teile der Arbeit, die mit den approximativen Verfahren enstanden sind – also die syntaktische und die semantische Analyse. Von ihnen hängt ab, inwieweit die generierten RFGs vollständig und korrekt sind. Im folgenden beschreibe ich zunächst nach welchen Kriterien ich die Ergebnisse beider Analysen betrachte und stelle dann die Ergebnisse vor.

Syntaxanalyse

Die Syntaxanalyse basiert auf den Grammatiken für Lexer und Parser, die ich in Kapitel 4 behandelt habe. Dort habe ich auch das testbasierte Verfahren beschrieben, das ich zu ihrer Erstellung angewendet habe. Es basiert auf der Eliminierung von Fehlern führt zu einer Syntaxanalse, die die verwendeten Beispielprogramme vollständig analysieren kann. Daher kann auch ohne weitere Untersuchung ausgesagt werden, dass die Grammatik die Beispielprogramme vollständig erkennt. Interessant ist hier eher welche Erkennungsleistung sie gegenüber anderen Programmen zeigt. Im der folgenden Betrachtung werde ich daher versuchen ein

weiteres Programm, das mit 30.000 Zeilen eine nennenswerte Größe erreicht (leider sind Programme, die dieses Kriterium erfüllen und zudem frei verfügbar sind, schwer zu finden) mit meiner Implementierung zu untersuchen. Das besagte Zusatzprogramm für den Test heißt *Sales and Inventory Manager*[1].

Namensauflösung

Der Kernpunkt der semantischen Analyse ist die Namensauflösung, ohne die eine korrekte Erkennung der Beziehungen der RFG-Entitäten nicht möglich wäre. In Abschnitt 4.3 habe ich beschrieben wie hierfür ein Modell entwickelt und implementiert wurde. Ein wesentlicher Punkt ist, dass auch hier die Grundlagen in einem Annäherungsverfahren entstanden sind. Daher gilt es zu prüfen, inwieweit dieses Modell und seine Umsetzung die Namensauflösung in Visual Basic 6 korrekt nachbilden.

Als Grundlage für die Untersuchung können die Fehler, die in der Namensauflösung auftreten verwendet werden. Um diese nachvollziehbar zu machen und zwischen Problemen, die durch das späte COM-Binden entstehen und tatsächlichen Fehlern des Algorithmus unterscheiden zu können, wurde in der Namensauflösung eine Methode implementiert, die ausschließlich dazu dient (qualifizierte) Bezeichner, deren Auflösung gescheitert ist, zu untersuchen.

Hierbei lassen sich die Probleme, die durch das späte Binden entstanden sind durch den Objekttyp erkennen, der dem nicht aufgelösten Qualifikationelement vorausgeht. Ist die Auflösung eines Elements gescheitert, weil das vorige von generischem Typ war (`Object`, `Variant` etc.), so ist dies dem Späten Binden zuzurechnen. Die Fehlerbehandlung kann daher das späte Binden erkennen und entsprechende Fehlermeldungen ausgeben.

Alle weiteren Fehler, die auftreten, können auf Fehler in der semantischen Analyse zurückgeführt werden, die Auftreten, weil das Modell für die Namensauflösung und die berücksichtigen Ausnahmen nicht vollständig sind oder weil ein Fehler in der Implementierung vorliegt.

Dabei ist anzumerken, dass diese Zahlen keinen direkten Aufschluss über falsche Positive liefern, also solche Fälle, in denen einem Bezeichner zwar ein Symbol zugeordnet wurde, dieses aber nicht das richtige war. Diese Fälle machen sich nur durch eventuell auftretenden Folgefehler oder die manuelle Überprüfung der Ergebnisse bemerkbar. Generell geht es hier aber darum grob abzuschätzen wie gut die Annährung an die Semantik von Visual Basic gelungen ist. Das Zählen der erfolreichen und erfolglosen Namensauflösungen liefert dazu vergleichbare Zahlenwerte.

Ergebnisse

Der Test der syntaktischen Analyse lieferte, wie erwartet, ein fehlerfreies Ergebnis für die Beispielprogramme. Beachtlich ist dagegen, dass auch das zusätzliche Programm analysiert werden konnte, nachdem lediglich ein Fehler im Designer-Code behoben werden musste – hier wurde eine Klammerung nicht erlaubt. Dies war das einzige Problem, dass beim Test auftrat.

Das Ergebnis der semantischen Analyse ist in 6.1 aufgeführt. Die dortigen Gesamtzahlen kennzeichnen die Anzahl der (eventuell qulifizierten) Bezeichner, die analysiert wurden. Tritt ein fehler bei der Auflösung eines solchen Bezeichners auf, so die gesamte Qualifizierung als Fehler gezählt.

Diese Werte zeigen, dass das gewählte Modell und seine Implementierung die allgemeinen

[1] Sales and Inventory Manager: `http://www.planetsourcecode.com/vb/scripts/ShowCode.asp?txtCodeId=66681&lngWId=1`

Testgegenstand	ERP-System		Micosoft-Beispiele	
Anzahl qual. Bezeichner	2.034.395	(100%)	20.956	(100%)
fehlerfreiaufgelöst	1.822.729	(89,6%)	19.938	(95,1%)
spät gebunden	211.666	(10,4%)	269	(1,3%)
unaufgelöste Sonderfälle	22.649	(1,1%)	749	(3,6%)

Tabelle 6.1: Testergebnisse für die semantische Analyse

Prinzipien der Namensauflösung in Visual Basic 6 offenbar korrekt abbilden und die Auflösung nur in Ausnahmefällen scheitert. Bereinigt man die Werte von den spät gebundenen Symbolen, so steigt beim ERP-System die Zahl der fehlerlosen Auflösung sogar auf 98,7%, bei den Microsoft-Beispielen auf 96,3%. In Anbetracht der schwierigen Ausgangslage und der approximativen Verfahren ist dies ein positives Ergebnis.

Testgegenstand	Sales and Inventory Manager	
Anzahl qual. Bezeichner)	17.269	(100%)
fehlerfreiaufgelöst	15158	(87,8%)
spät gebunden	45	(0,2%)
unaufgelöste Sonderfälle	2066	(10,3%)

Tabelle 6.2: Testergebnisse für die semantische Analyse

Bei der Analyse des zusätzlichen Programms, deren Ergebnisse in Tabelle 6.1 dargestellt werden wurde ebenfalls eine gute Erkennungsrate gemessen. Die Fehlerzahl ist jedoch höher. Wie sich bei einer näheren Analyse der Fehlermeldungen und der erzeugten RFGs zeigt, verhält sich die Implementierung bei der Analyse von bestimmten Modulen, die neue Oberflächenkomponenten definieren fehlerhaft. Der Grund hierfür liegt in den Beispielprogramme, die nur weniger solcher Module enthalten. Offenbar war die Menge der Beispielprogramme an dieser Stelle nicht repräsentativ genug.

Zwar ist das Ergebnis hier nicht so gut wie bei den Beispielprogrammen, dafür zeigt es aber eine weitere wichtige Eigenschaft der Analysewerkzeuge: Selbst bei einer größeren Anzahl von fehlenden Symbolen erfolgt die Analyse und RFG-Generierung, wobei unaufgelöste Beziehungen ausgelassen werden. Insofern verhält sich die Implementierung also fehlertolerant.

Bei den spät gebundenen Symbolen zeigt sich, dass ihre Zahl sich zwischen verschiedenen Softwaresystemen offenbar stark unterscheiden kann. Allerdings sind die Microsoft-Beispiele eine Sammlung eher kleinerer Anwendungen, während das ERP-System sehr umfangreich ist, über Objekte verfügt, die von unterschiedlichen Teilen der Implementierung genutzt werden und zudem recht stark modularisiert ist. Die wesentliche höhere Zahl an Aufrufen über das COM-Dispatching könnte daher rühren, dass die Systemkomponenten über COM miteinander in Verbindung stehen. Die Zahl von 10% ist höher als zu Beginn der Arbeit vermutet wurde.

6.2 Bewertung

Die angewendeten Methoden, die daraus resultierenden Grammatiken und Modell sowie deren Umsetzung erreichen das Ziel die Beispielprogramme zu analysieren. Zwar sind sie nicht vollständig, jedoch liegt es in der Natur eines Annährungsverfahrens, dass dieses vermultich nicht zu einem vollständigen und korrekten Ergebnis führt. Die vielen Besonderheiten, die in Visual Basic 6 stecken, erschweren das Vorgehen, da oft kein allgemeines Regelwerk zu erkennen ist.

Zwar lassen sich grundlegende Modelle erarbeiten, in jedem Fall muss aber eine Vielzahl von Ausnahmen davon berücksichtigt werden.

Wie obigen Ergebnisse zeigen, ist es mit den Grammatiken und dem Modell für die semantische Analyse gelungen die vorliegenden Beispielprogramme bis auf wenige Ausnahmen korrekt zu analyiesren. Vor dem Hindergund der schwierigen Ausgangslage sind diese Ergebnisse zufriendstellend. Auch die zusätzlich untersuchte Software ließ analysieren, wobei ebenfalls eine hohe Erkennungsrate erreicht wurde.

Wichtig ist letztendlich welchen Nutzen die generierten RFGs für den Anwender bieten. Die Tatsache, dass zum Teil Informationen fehlen, muss nicht bedeuten, dass Graphen ihren Nutzen verlieren. Beispielsweise ist die Herleitung der Architektur ein halbautomatischer Prozess, in dem die Analysen unterstützed fungieren. Sie stellen selbst keine exakten Methoden dar. Daher kann auch ein RFG, in dem einige Beziehungen fehlen – die Symbole des Programms sind ja im einzelnen bekannt, es fehlen lediglich Kanten – eine Grundlage zur Untersuchung der Software und ihrer Architektur bieten. Insofern sind die Ergebnisse dieser Arbeit für das Verstehen von Visual Basic Programmen von großem Wert.

6.3 Zusammenfassung

Die jüngste Entwicklung der Sprache Visual Basic 6 drängt die Entwickler, die sie einsetzen, dazu ihre bestehenden Softwaresysteme umzustellen. Zukünftig wird es keine offizielle Unterstützung seitens des Herstellers Microsoft geben. Langfristig ist damit zu rechnen, dass Programme, die auf dieser auslaufenden Sprache basieren auf neuen Betriebssystemen nicht mehr funktionsfähig sein werden. Die Umstellung auf die neue Version ist wegen der gravierenden Änderungen schwierig und erfordert, dass Teile der bestehenden Programme von Hand umgestellt werden. Dazu ist ein unfangreiches Verständnis der Software und ihres Aufbaus notwendig. Die Bauhaus-Suite hilft mit ihren Analysen und Darstellungen, dieses zu erlangen.

In dieser Arbeit habe ich die Möglichkeit geschaffen, die Bauhaus-Werkzeuge auch für Visual Basic 6 nutzen zu können, indem ich ein Frontend realisiert habe, mit dem der RFG für die Programme dieser Sprache erstellt werden kann. Zunächst habe ich hierzu die notwendigen Grundlagen erarbeitet und daraus ein Schema abgeleitet, das die Konzepte von Visual Basic 6 auf das RFG-Metamodell abbildet. Einige der Konzepte sind dabei erstmals auf den RFG abgebildet worden. Hierzu war es notwendig das Metamodell dahingehend erweitern, dass es Events und ihre Semantik darstellen kann. Für andere Programmelemente wie beispielsweise die Propertys war es notwendig Lösungen zu erarbeiten wie diese auf die vorhandenen RFG-Knoten und Kanten abgebildet werden können. Durch die Visualisierung von Attributen konnte dies ohne die Einführung weiterer neuer Konzepte erfolgen.

Die wesentliche Herausforderung bei der Analyse von Visual Basic 6 bestand in der unvollständigen Dokumentation und dem Fehlen einer Grammatikdefinition, die den Anforderungen der RFG-Generierung genügt. Ich habe daher verschiedene Vorgehensweisen betrachtet, die es ermöglichen mit dem partiellen Wissen umzugehen und ein Verfahren gewählt. Zur Rekonstruktion der Visual Basic Grammatik musste ich nach dem Versuch-und-Irrtum-Prinzip vorgehen, so wie es Lämmel und Verhoef in [LV01] vorstellen.

Für die semantische Analyse wurde ein grundlegendes Modell zur Namensauflösung aufgrund der Dokumentation, eigener Annahmen und Tests erstellt und implementiert. Auch hier war ein approximatives Vorgehen notwendig, um alle benötigten Informationen zusammenzutragen, da die verfügbare Dokumentation nicht ausreichte.

Ein generelles Problem stellten die vielen Besonderheiten von Visual Basic dar. Die Syntax der

Sprache bietet viele Sonderformen und beinhaltet viele Mehrdeutikeiten, was die Erstellung einer Grammatik erschwert. Ähnlich ist es bei der Semantik, die viele Aspekte bietet, die sich nicht in ein allgemeines Regelwerk zusammenfügen. Es sind die vielen Ausnahmen, die die Analyse der Sprache zu einer schwierigen Aufgabe machen. Wie sich in dieser Arbeit gezeigt hat lässt sich ein großer Teil der Semantik durch ein einheitliches Modell abbilden. Auch die Grammatik ist größtenteils nicht ungewöhnlich und wird erst durch die Sonderfälle kompliziert. In der begrenzten Zeit, die für diese Arbeit zur Verfügung stand konnten nicht alle dieser Sonderfälle behandelt werden. Die Annährung die geschaffen wurde, liefert jedoch, soweit möglich, nahezu vollständige Ergebnisse für die Beispielprogramme. Ein erster Test mit weiteren Programmen lieferte vielversprechende Ergebnisse.

Anhang

ANHANG A

Grammatiken

Dieser Anhang enthält die in dieser Arbeit erstellten Grammatiken in einer vereinfachten Form. Da alle Prädikate zur besseren Lesbarkeit ausgelassen wurden, sind sie nicht frei von Mehrdeutigkeiten. Zudem ist die Darstellung vereinheitlicht und daher nicht mit den Formulierungen des Originals identisch. Optionale Teile werden beispielsweise stets durch die Alternative des leeren Wortes anstatt durch die Notation „(...)?“ dargestellt.

Die hier dargestellte Version dient lediglich dazu einen Überblick über den allgemeinen Aufbau zu erhalten, da die vollständige Grammatik mit allen Prädikaten und Actions weniger leicht zu lesen ist. Sowohl die vollständige Grammatikdefinition, als auch eine HTML-Variante der hier abgebildeten Form sind in der digitalen Abgabe enthalten.

Im folgenden werden für den Visual Basic 6 und das Austauschformat für die Schnittstellen von COM-Bibliotheken aufgeführt. Hierbei wird jeweils zuerst die Lexer-Grammatik (Die Lexer haben Grammatiken da sie den $LL(k)$-Algorithmus verwenden , siehe Abschnitt 4.1.2.1) und dann die des Parsers dagestellt.

A.1 Visual Basic 6 – Lexer

```
mWS
    :   ( ' ' | '\t' )+
    ;

mEOL
    :   ( '\n' | "\r\n" | '\r' )
    ;

mIDENTIFIER
    :       ( mLETTER | '_' (   mLETTER | mDIGIT ) )
            ( mLETTER | mDIGIT | '_' )*
            mCOLON
        |   mCOMMENT_REM
        |   ( mLETTER | ( '_' ) ( mLETTER | mDIGIT ) )
            ( mLETTER | mDIGIT | '_' )*
    ;

protected mLETTER
    :   'a'..'z' |  'ä' |  'ö' |  'ü' |  'ß'
    ;
```

```
protected mDIGIT
    :   '0'..'9'
    ;

mCOLON
    :   ':'
    ;

protected mCOMMENT_REM
    :   "rem" mCOMMENT_BODY
    ;

protected mFILE_IDENTIFIER
    :   '#' ( mLETTER | mDIGIT | '_' )+
    ;

mFILE_IDENT_OR_DATE
    :   mDATE_LITERAL | mFILE_IDENTIFIER
    ;

protected mDATE_LITERAL
    :   '#' ( mDATE ( mWS mTIME | /* leeres Wort */ ) | mTIME ) '#'
    ;

mPLUS
    :   '+'
    ;

mPLUS_ASSIGN
    :   "+="
    ;

mMINUS
    :   '-'
    ;

mMINUS_ASSIGN
    :   "-="
    ;

mSTAR
    :   '*'
    ;

mSTAR_ASSIGN
    :   "*="
    ;

mDIV
    :   '/'
```

```
    ;

mDIV_ASSIGN
    :   "/="
    ;

mINT_DIV
    :   '\\'
    ;

mEXPO
    :   '^'
    ;

mEQUAL
    :   '='
    ;

mNOT_EQUAL
    :   "<>"
    ;

mLESS_THAN
    :   '<'
    ;

mLESS_EQUAL
    :   "<="
    ;

mGREATER_THAN
    :   '>'
    ;

mGREATER_EQUAL
    :   ">="
    ;

mAMP
    :   '&'
    ;

mCOLON_EQUALS
    :   ":="
    ;

mTS_INT
    :   '%'
    ;
```

```
mTS_DOUBLE
    :   '#'
    ;

mTS_CURRENCY
    :   '@'
    ;

mTS_STRING
    :   '$'
    ;

mBANG_OR_TS_SINGLE
    :   '!' | '!'
    ;

protected mLBRACE
    :   '['
    ;

protected mTYPE_LITERAL
    :   mINT_TYPE_LITERAL | mFLOAT_TYPE_LITERAL | mTS_STRING
    ;

protected mINT_TYPE_LITERAL
    :   mTS_INT | mAMP
    ;

protected mFLOAT_TYPE_LITERAL
    :   mBANG_OR_TS_SINGLE | mTS_DOUBLE | mTS_CURRENCY
    ;

mCOMMA
    :   ','
    ;

mSEMI
    :   ';'
    ;

mLPAREN
    :   '('
    ;

mRPAREN
    :   ')'
    ;

mLCURLY
    :   '{'
```

```
    ;

mRCURLY
    :   '}'
    ;

mDOT
    :   '.'
    ;

protected mRBRACE
    :   ']'
    ;

mESCAPED_NAME
    :   mLBRACE ( ~']' )* mRBRACE
    ;

mST_CONT
    :   '_' mEOL
    ;

mSTRING_LITERAL
    :   '"'
            (
                ( '"' | '\n' | '\r')
                | "\"\""
            )*
        '"'
    ;

mNUM_INT
    :       '1'..'9' ( mDIGIT )*
            (   'e' ( mPLUS | mMINUS | /* leeres Wort */ ) ( mDIGIT )+
            |   /* leeres Wort */
            )
            (   mTYPE_LITERAL | mFRACTION | /* leeres Wort */ )
        |   '0' ( mTYPE_LITERAL | mFRACTION | /* leeres Wort */ )
    ;

protected mFRACTION
    :   '.' ( mDIGIT )+
        (   'e' ( mPLUS | mMINUS | /* leeres Wort */ ) ( mDIGIT )+
        |   /* leeres Wort */
        )
        (   mFLOAT_TYPE_LITERAL | /* leeres Wort */ )
    ;

mNUM_HEX
    :   '&' 'h'
```

```
        ( mDIGIT | 'a'..'f' )+
        ( mINT_TYPE_LITERAL | /* leeres Wort */ )
    ;

mNUM_OCT
    :   '&' 'o' ( '0'..'7' )+ ( mINT_TYPE_LITERAL | /* leeres Wort */ )
    ;

protected mHEX_DIGIT
    :   '0'..'9' | 'a'..'f'
    ;

protected mHEX_LITERAL
    :   '&' 'h' ( mHEX_DIGIT )+
    ;

protected mOCT_DIGIT
    :   '0'..'7'
    ;

protected mOCT_LITERAL
    :   '&' 'o' ( mOCT_DIGIT )+
    ;

mHEX_OFFSET
    :   mCOLON mHEX_DIGIT mHEX_DIGIT mHEX_DIGIT mHEX_DIGIT ( mHEX_DIGIT )*
    ;

protected mDATE
    :   ( mDIGIT )+ '/' ( mDIGIT )+ '/' ( mDIGIT )+
    ;

protected mTIME
    :   ( mDIGIT )+ ':' ( mDIGIT )+
        ( ':' ( mDIGIT )+ | /* leeres Wort */ )
        mWS
        ( "am" | "pm" )
    ;

mKEY_LITERAL
    :   mLCURLY
        ( mDIGIT | mLETTER | mMINUS )+
        mRCURLY
    ;

mCOMMENT
    :   '\'' mCOMMENT_BODY
    ;

protected mCOMMENT_BODY
```

```
:   ( mST_CONT | ( '\n' | '\r' ) )*
;
```

A.2 Visual Basic 6 –Parser

```
compilation_unit
    :   ( form_module | class_module | standard_module )
    ;

form_header
    :   "version" NUM_FLOAT EOL
    ;

form_module
    :   form_header form_com_object_def form_designer_code
        module_header_attributes module_header_options module_body
    ;

class_module
    :   class_header class_attributes module_header_attributes
        module_header_options module_body
    ;

standard_module
    :   module_header_attributes module_header_options module_body
    ;

form_com_object_def
    :   (   "object" EQUAL STRING_LITERAL
            ( SEMI STRING_LITERAL | /* leeres Wort */)
            EOL
        )*
    ;

form_designer_code
    :   ( form_designer_code_block )*
    ;

module_header_attributes
    :   ( module_header_attribute_decl )*
    ;

module_header_options
    :   ( module_header_option_decl )*
    ;

module_body
    :   ( module_body_statement )*
    ;

form_designer_code_block
    :   "begin" type_identifier member_identifier EOL
        (   form_designer_assignment
```

```
        |   form_designer_code_block
        |   form_designer_property_block
        )*
        "end" EOL
    ;

type_identifier
    :   id
    ;


member_identifier
    :   var_identifier
        |   (   "alias"
            |   "append"
            |   "end"
            |   "endproperty"
            |   "enum"
            |   "event"
            |   "function"
            |   "let"
            |   "new"
            |   "next"
            |   "open"
            |   "optional"
            |   "output"
            |   "select"
            |   "set"
            |   "sub"
            |   "to"
            |   "type"
            |   "typeof"
            |   "rem"
            |   "version"
            )
            (   type_shortcut
            |   /* leeres Wort */
            )
    ;


form_designer_assignment
    :   member_identifier ( identifier_suffix )* EQUAL form_designer_constant
        EOL
    ;

form_designer_property_block
    :   "beginproperty" member_identifier
        ( LPAREN argument_list RPAREN | /* leeres Wort */ )
        ( KEY_LITERAL | /* leeres Wort */   )
```

```
        EOL
        ( form_designer_assignment
        |   (   form_designer_coll_property_block
            |   form_designer_property_block
            )
        )*
        "endproperty" EOL
    ;

argument_list
    :   argument ( COMMA argument )*
    ;

form_designer_coll_property_block
    :   "beginproperty" member_identifier
        ( LPAREN argument_list RPAREN | /* leeres Wort */ )
        ( KEY_LITERAL | /* leeres Wort */ )
        EOL
        ( form_designer_assignment | form_designer_property_block )*
        "endproperty" EOL
    ;

identifier_suffix
    :       ( DOT | BANG )
            member_identifier
        |   bracketed_argument_list
    ;

form_designer_constant
    :   (   ( MINUS | /* leeres Wort */)
            (   /* leeres Wort */
            |   NUM_INT ( COMMA ( NUM_INT )+ | /* leeres Wort */ )
            |   NUM_FLOAT
            |   NUM_LONG
            |   NUM_DOUBLE
            |   NUM_SINGLE
            |   NUM_CURRENCY
            )
        )
        |   STRING_LITERAL
        |   CHAR_LITERAL
        |   ( PLUS | EXPO )* ( KEY_LITERAL | IDENTIFIER )
        |   FRX_LITERAL
        |   "true"
        |   "false"
        |   "nothing"
    ;

class_header
    :   "version" NUM_FLOAT "class" EOL
```

```
    ;

class_attributes
    :   "begin" EOL ( IDENTIFIER EQUAL expression EOL )* "end" EOL
    ;

expression
    :   impExpression
    ;

module_header_attribute_decl
    :   "attribute"
        (   IDENTIFIER EQUAL STRING_LITERAL eos
        |   IDENTIFIER EQUAL ( "true" | "false" ) eos
        |   IDENTIFIER EQUAL constant ( COMMA constant )* eos
        )
    ;

eos
    :   ( EOL | COLON )+
    ;

constant
    :   (   NUM_INT
        |   CHAR_LITERAL
        |   STRING_LITERAL
        |   NUM_FLOAT
        |   NUM_LONG
        |   NUM_DOUBLE
        |   NUM_SINGLE
        |   NUM_CURRENCY
        |   "true"
        |   "false"
        |   "nothing"
        |   "null"
        |   "empty"
        |   KEY_LITERAL
        |   DATE_LITERAL
        )
    ;

module_header_option_decl
    :   "option"
        (   "base" NUM_INT
        |   "compare" ( "binary" | "text" | "database" )
        |   "explicit"
        |   "private" "module"
        )
        eos
    ;
```

```
module_body_statement
    :       modifiers
            (   sub
            |   function
            |   property
            |   enumeration
            |   type
            |   event
            |   declare_statement
            |   module_body_var_decl
            |   module_body_const_decl
            )
        |   deftype_statement
        |   "implements" identifier eos
    ;

modifiers
    :   ( modifier )*
    ;

sub
    :   "sub" var_identifier
        ( LPAREN ( parameter_list | /* leeres Wort */ ) RPAREN
        | /* leeres Wort */
        )
        eos
        ( statement_block | /* leeres Wort */ )
        END_SUB eos
    ;


function
    :   "function" var_identifier
        ( LPAREN ( parameter_list | /* leeres Wort */ ) RPAREN
        | /* leeres Wort */
        )
        ( "as" type_spec | /* leeres Wort */ )
        eos
        ( statement_block | /* leeres Wort */ )
        END_FUNCTION eos
    ;

property
    :   "property" ( "get" | "let" | "set" )
        var_identifier
        ( LPAREN ( parameter_list | /* leeres Wort */ ) RPAREN
        | /* leeres Wort */
        )
        ( "as" type_spec | /* leeres Wort */ )
```

```
        eos
        ( statement_block | /* leeres Wort */ )
        END_PROPERTY eos
    ;

enumeration
    :   "enum" var_identifier eos enumeration_member_list END_ENUM eos
    ;

type
    :   "type" var_identifier eos type_member_list END_TYPE eos
    ;

event
    :   "event" IDENTIFIER
        ( LPAREN ( parameter_list | /* leeres Wort */ ) RPAREN
        | /* leeres Wort */
        )
        eos
    ;

declare_statement
    :   "declare"
        ( "sub" | "function" )
        var_identifier "lib" STRING_LITERAL
        ( "alias" STRING_LITERAL | /* leeres Wort */ )
        ( LPAREN ( parameter_list | /* leeres Wort */ ) RPAREN
        | /* leeres Wort */
        )
        ( "as" type_spec | /* leeres Wort */ )
        eos
    ;

module_body_var_decl
    :   module_body_var_decl_body ( COMMA module_body_var_decl_body )* eos
    ;

module_body_const_decl
    :   const_statement eos
    ;

deftype_statement
    :   (   "defbool"
        |   "defbyte"
        |   "defint"
        |   "deflng"
        |   "defcur"
        |   "defsng"
        |   "defdbl"
        |   "defdec"
```

```
        |   "defdate"
        |   "defstr"
        |   "defobj"
        |   "defvar"
        )
        deftype_letterrange ( COMMA deftype_letterrange )* eos
    ;

identifier
    :   id
    ;

const_statement
    :   "const" const_statement_var_decl ( COMMA const_statement_var_decl )*
    ;

module_body_var_decl_body
    :   ( "dim" | /* leeres Wort */ )
        ( "withevents"  | /* leeres Wort */ )
        var_identifier
        ( array_decl_suffix | /* leeres Wort */ )
        ( "as" ( "new" | /* leeres Wort */ ) type_spec | /* leeres Wort */ )
    ;

var_identifier
    :   (   IDENTIFIER
        |   FILE_IDENTIFIER
        |   ESCAPED_NAME
        |   "base"
        |   "class"
        |   "database"
        |   "date"
        |   "error"
        |   "get"
        |   "input"
        |   "len"
        |   "lib"
        |   "line"
        |   "name"
        |   "object"
        |   "print"
        |   "property"
        |   "step"
        |   "string"
        |   "text"
        |   "time"
        |   "write"
        )
        (   type_shortcut  | /* leeres Wort */ )
    ;
```

```
array_decl_suffix
    :   LPAREN ( array_size_spec_list | /* leeres Wort */ ) RPAREN
    ;

type_spec
    :       builtin_type
        |   "string" STAR ( constant | identifier )
        |   type_identifier
    ;

parameter_list
    :   parameter ( COMMA parameter )*
    ;

statement_block
    :   ( terminated_statement )+
    ;

enumeration_member_list
    :   ( enumeration_member )+
    ;

enumeration_member
    :   member_identifier
        ( "as" type_spec | /* leeres Wort */ )
        ( EQUAL expression | /* leeres Wort */ )
        eos
    ;

type_member_list
    :   ( type_member )+
    ;

type_member
    :   member_identifier
        ( array_decl_suffix | /* leeres Wort */ )
        ( "as" type_spec | /* leeres Wort */ )
        eos
    ;

parameter
    :   ( "optional" | /* leeres Wort */ )
        ( "byref" | "byval" | /* leeres Wort */ )
        ( "paramarray" | /* leeres Wort */ )
        member_identifier
        ( array_decl_suffix | /* leeres Wort */ )
        ( "as" type_spec | /* leeres Wort */ )
        ( EQUAL expression | /* leeres Wort */ )
    ;
```

```
deftype_letterrange
    :   IDENTIFIER ( MINUS IDENTIFIER | /* leeres Wort */ )
    ;

terminated_statement
    :       statement eos
        |   ( LINE_NUMBER | LINE_LABEL )
        |   ( LINE_NUMBER | LINE_LABEL ) eos
    ;


statement
    :       legacy_statement
        |   assignment
        |   simple_call_statement
        |   dim_statement
        |   redim_statement
        |   procedue_const_statement
        |   static_statement
        |   attribute_statement
        |   "end"
        |   "set" assignment
        |   if_statement
        |   "for" identifier EQUAL expression "to" expression
            ( "step" expression | /* leeres Wort */ )
            eos statement_block NEXT
            ( var_identifier | /* leeres Wort */ )
        |   "for" "each" identifier "in" identifier eos statement_block NEXT
            ( var_identifier | /* leeres Wort */ )
        |   do_loop_statement
        |   "while" expression eos
            ( statement_block | /* leeres Wort */)
            "wend"
        |   "with" identifier eos
            ( statement_block | /* leeres Wort */ )
            END_WITH
        |   select_case_statement
        |   "exit"
            ( "do" | "for" | "function" | "property" | "sub" )
        |   "on"
            ( "local" | /* leeres Wort */ )
            "error"
            ( "goto" ( label_identifier | NUM_INT )
            | "resume" "next"
            )
        |   "raiseevent" event_identifier
        |   "resume"
            ( "next" | NUM_INT | label_identifier | /* leeres Wort */ )
        |   goto_statement
```

```
        |   "return"
    ;

legacy_statement_head
    :       ( "date" | "time" ) EQUAL
        |   ( "name" | "open" ) expression
        |   ( "lock" | "unlock" | "print" ) identifier
        |   "line" "input"
    ;

legacy_statement
    :   "date" EQUAL expression
        |   "name" expression "as" expression
        |   "line" "input" identifier COMMA identifier
        |   ( "lock" | "unlock" ) identifier
            ( NUM_INT ( "to" NUM_INT | /* leeres Wort */ )
            | /* leeres Wort */
            )
        |   "open" expression "for"
            ( "append" | "binary" | "input" | "output" | "random" )
            ( "access"
                ( "read" ( "write" | /* leeres Wort */ )
                | "write"
                )
            | /* leeres Wort */
            )
            ( "shared"
            | "lock"
              ( "read" ( "write" | /* leeres Wort */ )
              | "write"
              | /* leeres Wort */
              )
            | /* leeres Wort */
            )
            "as" identifier ( "len" EQUAL NUM_INT | /* leeres Wort */)
        |   "time" EQUAL expression
    ;

assignment_header
    :   ( "let" | /* leeres Wort */ )
        identifier
        ( EQUAL | PLUS_ASSIGN | MINUS_ASSIGN | STAR_ASSIGN | DIV_ASSIGN )
    ;

assignment
    :   assignment_header expression
    ;

simple_call_statement
    :   ( "call" | /* leeres Wort */ )
```

```
        ( var_identifier | ( UNARY_DOT | UNARY_BANG ) member_identifier )
        ( ( DOT | BANG ) member_identifier | bracketed_argument_list )*
        ( argument_list | drawing_method_call | print_method_call )
    ;

dim_statement
    :   "dim" dim_var_decl ( COMMA dim_var_decl )*
    ;

redim_statement
    :   "redim" ( "preserve" | /* leeres Wort */ )
        redim_decl ( COMMA redim_decl )*
    ;

procedue_const_statement
    :   const_statement
    ;

static_statement
    :   "static" static_var_decl ( COMMA static_var_decl )*
    ;

attribute_statement
    :   "attribute" ( IDENTIFIER DOT IDENTIFIER EQUAL NUM_INT | assignment )
    ;

if_statement
    :   "if" expression "then"  ( inline_if_body | multiline_if_body )
    ;

do_loop_statement
    :       do_cond_loop_head "loop"
        |   do_no_cond_loop_head "loop"
            ( ( "while" | "until" ) expression | /* leeres Wort */ )
    ;

select_case_statement
    :   "select" "case" expression eos
        ( "case" select_case_condition eos
            ( statement_block | /* leeres Wort */ )
        )*
        ( "case" "else" eos ( statement_block | /* leeres Wort */ )
        | /* leeres Wort */
        )
        END_SELECT
    ;

label_identifier
    :   id
    ;
```

```
event_identifier
    :   id
    ;

goto_statement
    :   goto_body
    |   "on" expression goto_body ( COMMA ( label_identifier | NUM_INT ) )*
    ;

inline_if_body
    :   ( COLON | /* leeres Wort */ )
        statement ( COLON statement )*
        ( "else" statement ( COLON statement )* | /* leeres Wort */ )
    ;

multiline_if_body
    :   eos
        ( statement_block | /* leeres Wort */ )
        ( "elseif" expression "then"
            ( statement | /* leeres Wort */ )
            eos
            ( statement_block | /* leeres Wort */ )
        )*
        ( "else"
          ( statement | /* leeres Wort */ )
          eos
          ( statement_block |/* leeres Wort */ )
        | /* leeres Wort */
        )
        END_IF
    ;

select_case_condition
    :       select_case_condition_expression
            ( COMMA select_case_condition_expression )*
        |   IS
            ( EQUAL | LESS_EQUAL | LESS_THAN | GREATER_EQUAL | GREATER_THAN )
            expression
    ;

select_case_condition_expression
    :   expression
        ( "to" expression | /* leeres Wort */ )
    ;

dim_var_decl
    :   ( "withevents" | /* leeres Wort */ )
        var_identifier
        ( array_decl_suffix | /* leeres Wort */ )
```

```
        ( "as" ( "new" | /* leeres Wort */ ) type_spec
        | /* leeres Wort */
        )
    ;

const_statement_var_decl
    :   member_identifier
        ( "as" type_spec | /* leeres Wort */ )
        EQUAL expression
    ;

redim_decl
    :   identifier ( "as" type_spec | /* leeres Wort */ )
    ;

static_var_decl
    :   var_identifier
        ( array_decl_suffix | /* leeres Wort */ )
        ( "as" ( "new" | /* leeres Wort */ ) type_spec
        | /* leeres Wort */
        )
    ;

do_cond_loop_head
    :   "do" ( "while" | "until" ) expression eos statement_block
    ;

do_no_cond_loop_head
    :   "do" eos statement_block
    ;

goto_body
    :   ( "goto" | "gosub" ) ( label_identifier | NUM_INT )
    ;

bracketed_argument_list
    :   LPAREN argument_list RPAREN
    ;

drawing_method_call
    :   drawing_method_call_point_def
        ( drawing_method_call_point_def | /* leeres Wort */ )
        ( COMMA argument_list | /* leeres Wort */)
    ;

print_method_call
    :   expression_list
        ( SEMI | SEMI expression_list )*
    ;
```

```
drawing_method_call_point_def
    :   ( MINUS | /* leeres Wort */ )
        ( "step" | /* leeres Wort */ )
        LPAREN expression COMMA expression RPAREN
    ;

expression_list
    :   expression ( COMMA expression )*
    ;

impExpression
    :   eqvExpression ( IMP eqvExpression )*
    ;

eqvExpression
    :   xorExpression ( EQV xorExpression )*
    ;

xorExpression
    :   orExpression ( XOR orExpression )*
    ;

orExpression
    :   andExpression ( OR andExpression )*
    ;

andExpression
    :   notExpression ( AND notExpression )*
    ;

notExpression
    :   NOT notExpression | comparisonExpression
    ;

comparisonExpression
    :   stringConcatExpression (
            (   EQUAL
            |   NOT_EQUAL
            |   LESS_THAN
            |   GREATER_THAN
            |   LESS_EQUAL
            |   GREATER_EQUAL
            |   IS
            |   LIKE
            )
            stringConcatExpression )*
    ;

stringConcatExpression
    :   additiveExpression ( AMP additiveExpression )*
```

```
    ;


additiveExpression
    :   modExpression ( ( PLUS | MINUS ) modExpression )*
    ;

modExpression
    :   intDivExpression ( MOD intDivExpression )*
    ;

intDivExpression
    :   multiplicativeExpression ( INT_DIV multiplicativeExpression )*
    ;

multiplicativeExpression
    :   unaryExpression ( ( STAR | DIV | EXPO ) unaryExpression )*
    ;

unaryExpression
    :       PLUS unaryExpression
        |   MINUS unaryExpression
        |   EXPO unaryExpression
        |   "addressof" unaryExpression
        |   "typeof" unaryExpression
        |   primaryExpression
        |   "new" type_identifier
    ;

primaryExpression
    :   constant | identifier | LPAREN expression RPAREN
    ;

id
    :   (   var_identifier ( identifier_suffix )*
        |   ( UNARY_DOT | UNARY_BANG ) member_identifier ( identifier_suffix )*
        )
    ;

named_arg_identifier
    :   id
    ;

type_shortcut
    :   TS_CURRENCY | TS_DOUBLE | TS_INT | TS_STRING | TS_LONG | TS_SINGLE
    ;

argument
    :       named_arg_identifier COLON_EQUALS expression
        |   ( "byval" | "byref" | /* leeres Wort */ )
```

```
            expression
            ( "to" expression
            | /* leeres Wort */
            )
        |   /* leeres Wort */
    ;

builtin_type
    :       "byte"
        |   "boolean"
        |   "integer"
        |   "long"
        |   "single"
        |   "double"
        |   "currency"
        |   "variant"
    ;

modifier
    :   "public" | "private" | "friend" | "global" | "static"


array_size_spec_list
    :   array_size_spec ( COMMA array_size_spec )*
    ;

array_size_spec
    :   expression ( "to" expression | /* leeres Wort */ )
    ;
```

A.3 Austauschformat für COM-Schnittstellen – Lexer

```
mWS
    :   ( ' ' | '\t' )+
    ;

mEOL
    :   ( '\n' | "\r\n" | '\r' )+
    ;

mIDENTIFIER
    :   ( mLETTER | mDIGIT | '_' | '-' | '.' )+
    ;

protected mLETTER
    :   'a'..'z' | 'ä' |   'ö' | 'ü' | 'ß'
    ;

protected mDIGIT
```

```
    :   '0'..'9'
    ;

protected mHEX_DIGIT
    :   '0'..'9' | 'a'..'f'
    ;

mCOMMA
    :   ','
    ;

mLPAREN
    :   '('
    ;

mRPAREN
    :   ')'
    ;

mLTHAN
    :   '<'
    ;

mGTHAN
    :   '>'
    ;

mQUESTION_MARK
    :   '?'
    ;

mCOLON
    :   ':'
    ;

mGUID
    :   '{' ( mHEX_DIGIT | '-' )+ '}'
    ;

mCOMMENT
    :   '/' '/' ( ( '\n' | '\r' ) )*
    ;
```

A.4 Austauschformat für COM-Schnittstellen – Parser

```
lib_spec
: "library" identifier EOL
( "oca" GUID EOL | /* leeres Wort */ )
"guid" GUID EOL ( type_decl )*
```

```
;

identifier
: IDENTIFIER
| "optional"
| "alias"
| "as"
| "library"
| "guid"
| "global"
| "sub"
| "function"
| "property"
| "const"
| "typedef"
| "end"
| "class"
| "coclass"
| "module"
| "enum"
| "type"
| "appobject"
| "control"
| "licensed"
| "predeclid"
| "dual"
| "dispatchable"
| "nonextensible"
| "noncreatable"
| "hidden"
| "event"
| "aggregatable"
;


type_decl
: ( (  LTHAN "global" GTHAN EOL
( member_decl )*
"end" LTHAN "global" GTHAN EOL
)
| ( ( "class" | "coclass" | "module" | "enum" | "type" )
identifier
( "alias" identifier | /* leeres Wort */ )
( GUID | /* leeres Wort */ )
( COLON type_attribute_list | /* leeres Wort */ )
EOL ( member_decl )* "end"
( "class" | "coclass" | "module" | "enum" | "type" )
EOL
)
| ( "typedef" identifier EOL )
```

```
)

;

member_decl
: (  "const" identifier ( member_type  | /* leeres Wort */ ) EOL
| "property" identifier
( LPAREN ( parameter_list | /* leeres Wort */ ) RPAREN
| /* leeres Wort */
)
( member_type
| /* leeres Wort */
)
EOL
| "function" identifier
( LPAREN ( parameter_list | /* leeres Wort */ ) RPAREN
| /* leeres Wort */
)
( member_type
| /* leeres Wort */
)
EOL
| "sub" identifier
( LPAREN ( parameter_list | /* leeres Wort */ ) RPAREN
| /* leeres Wort */
)
EOL
| "event" identifier
( LPAREN  ( parameter_list | /* leeres Wort */ ) RPAREN
| /* leeres Wort */
)
EOL
)

;

type_attribute_list
: type_attribute ( COMMA type_attribute )*
;

type_attribute
: "appobject"
| "control"
| "licensed"
| "predeclid"
| "dual"
| "dispatchable"
| "nonextensible"
| "noncreatable"
| "hidden"
```

```
| "aggregatable"
;


member_type
: "as" ( identifier | QUESTION_MARK )
;

parameter_list
: parameter_decl ( COMMA parameter_decl )*
;

parameter_decl
: ( "optional" | /* leeres Wort */ )
( "byval" | "byref" | /* leeres Wort */ )
( "paramarray" | /* leeres Wort */ )
identifier
( LPAREN RPAREN | /* leeres Wort */ )
( member_type | /* leeres Wort */ )
;
```

ABBILDUNGSVERZEICHNIS

LISTINGS

LITERATURVERZEICHNIS

[ASU86] Aho, Alfred V., Ravi Sethi und Jeffrey D. Ullman: *Compilers: Principles, Techniques and Tools.* Prentice Hall, 1986.

[Axi06] Axivion: *Language Guide 5.3.* Axivion GmbH, 2006. RFG-Rerferenz.

[Cag06] Cager, Paul: *Java Parser for VB (Visual Basic).* Webseite, Stand: November 2006. `http://www.paulcager.org/products/vbparser/`.

[Cow00] Cowell, John: *Essential Visual Basic 6 fast.* Springer, 2000.

[Dud01] Duden: *Duden Informatik.* Dudenverlag, dritte Auflage, 2001.

[EKP+99] Eisenbarth, Thomas, Rainer Koschke, Erhard Plödereder, Jean-François Girard und Martin Würthner: *Projekt Bauhaus: Interaktive und inkrementelle Wiedergewinnung von SW-Architekturen.* In: *Workshop Software-Reengineering, Bad Honnef, Universität Koblenz-Landau, Fachberichte Informatik, Nr. 7-99*, Seiten 17–26, 1999.

[FP96] Fenton, Norman E. und Shari Lawrence Pfleeger: *Software Metrics: A Rigorous & Practical Approach.* Thomson Computer Press, Zweite Auflage, 1996.

[GHJV94] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[Hau99] Haupt, Horst F.: *Visual Basic 6.0 Referenz.* Franzis-Verlag, 1999.

[HU79] Hopcroft, John E. und Jeffrey D. Ullman: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie.* Addison-Wesley, 1979.

[ISO96] *Information technology – Syntactic metalanguage – Extended BNF.* Standard. ISO/IEC 14977, International Organization for Standardization, 1996.

[Kof00] Kofler, Michael: *Visual Basic 6: Programmiertechniken, Datenbanken, Internet.* Addison-Wesley-Longman, 2000.

[Loo01] Loos, Peter: *Go To COM: Das Objektmodell im Detail betrachtet, COM von Grund auf Beispielorientiert.* Addison-Wesley, 2001.

[LTP04] Lethbridge, Timothy, Sander Tichelaar und Erhard Plödereder: *The Dagstuhl Middle Metamodel: A Schema For Reverse Engineering.* Electronic Notes in Theoretical Computer Science, Vol. 94:7 – 18, Juli 2004.

[LV01] Lämmel, Ralf und Chris Verhoef: *Semi-automatic Grammar Recovery.* Software - Practice and Experience, Vol. 31(15):1395–1438, Dezember 2001.

[Mas99] Maslo, Andreas: *Das große Buch: Visual Basic 6.* Data Becker, 1999.

[Mic92] Microsoft: *Microsoft P-Code Technology.* Webseite, April 1992. http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/dnarvc/html/msdn_c7pcode2.asp, Stand: November 2006.

[Mic03] Microsoft: *Late, ID, Early Binding Types Possible in VB for Apps.* Webseite, Dezember 2003. Support-Artikel, http://support.microsoft.com/?scid=kb%3Ben-us%3B138138&x=8&y=7, Stand: November 2006.

[Mic06a] Microsoft: *The Component Object Model.* Webseite, Stand: November 2006. http://msdn2.microsoft.com/en-gb/library/ms694363.aspx.

[Mic06b] Microsoft: *Microsoft Developer Network Library - Visual Basic 6.* Webseite, 10.06. 2006. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnanchor/html/vb6anchor.asp.

[Mic06c] Microsoft: *Support Statement for Visual Basic 6.0 on Windows Vista.* Webseite, stand: Oktober 2006. http://msdn2.microsoft.com/en-us/vbrun/ms788708.aspx.

[Mic06d] Microsoft: *Upgrading Visual Basic 6.0 Applications to Visual Basic .NET and Visual Basic 2005.* Webseite, Stand: November 2006. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html/vb6tovbnetupgrade.asp.

[Mic06e] Microsoft: *Übersicht Microsoft Support Lifecycle.* Webseite, stand: Oktober 2006. http://support.microsoft.com/default.aspx?scid=fh;[ln];lifecycle.

[MN95] Murphy, Gail C. und David Notkin: *Lightweight Source Model Extraction.* In: *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, Seiten 116–127. ACM Press, 1995.

[Mon01] Monadjemi, Peter: *Visual Basic 6 Kompendium.* Markt+Technik, 2001.

[Moo02] Moonen, Leon: *Exploring Software Systems.* Doktorarbeit, Universität Amsterdam, Fakultät für Naturwissenschaften, Mathematik und Informatik, 2002.

[Mü] Müller, Markus: *Konzeption und Generierung eines RFG für COBOL.* Studienarbeit Nr. 1915, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Institut für Softwaretechnologie (ISTE), Februar 2004.

[PQ95] Parr, Terence und R. W. Quong: *ANTLR: A Predicated-LL(k) Parser Generator.* Software - Practice and Experience, Vol. 25(7), Juli 1995.

[RJB99] Rumbaugh, James, Ivar Jacobson und Grady Booch: *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.

[RVP06] Raza, Aoun, Gunther Vogel und Erhard Plödereder: *Bauhaus – a Tool Suite for Program Analysis and Reverse Engeneering.* In: *Reliable Software Technologies – Ada-Europe 2006*, Seiten 71–82, Juni 2006.

[Sch98] Schiffer, Stefan: *Visuelle Programmierung.* Addison-Wesley, 1998.

[Som04] Sommerville, Ian: *Software Engineering.* Addison-Wesley, achte Auflage, 2004.

[Tic01] Tichelaar, Sander: *Modeling Object-Oriented Software for Reverse Engineering and Refactoring.* Doktorarbeit, Universität Bern, Philosophisch-naturwissenschaftliche Fakultät, Dezember 2001.

[Vic03] Vick, Paul: *Dictionary Lookup Operator*. Blog, 15.07. 2003. `http://www.panopticoncentral.net/archive/2003/07/15/153.aspx`.

[Wil99] Wilson, Robert Paul: *Efficient, Context-Sensitive Pointer Analysis for C Programs*. Doktorarbeit, Stanford University, Depatment of Electrical Engineering, Dezember 1999.

[WM96] Watson, Arthur H. und Thomas J. McCabe: *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*. Technischer Bericht NIST Special Publication 500-235, National Institute of Standards and Technology, August 1996.