8 Evaluation

Dieses Kapitel soll die entworfenen Konzepte, und vor allem deren implementierte Teile, einer kritischen Bewertung unterziehen. Hierzu wird zunächst die Verwendbarkeit der Komponenten-Bibliothek anhand einiger kleiner, exemplarischer Modellgeneratoren aufgezeigt. Anschließend werden die Entwürfe und Implementationen im Rahmen der in Kapitel 5 aufgeführten Bewertungskriterien evaluiert.

8.1 Beispiele für Modellgeneratoren

Die folgenden Modellgeneratoren sollen zeigen, wie die COM-Modellierungsbibliothek vorteilhaft eingesetzt werden kann. Die Codebeispiele beziehen sich auf kleinere Modelle aus der Literatur und sind in VB.NET[1] geschrieben.

8.1.1 Einfaches Produktionsmodell

Das folgende triviale Produktionsmodell mit zwei Gütern *XB* und *XC* entstammt aus [Fourer03, S. 5] und dient dort als Einführungsbeispiel für AMPL.

```
01: var XB;
02: var XC;
03: maximize Profit: 25 * XB + 30 * XC;
04: subject to Time: (1/200) * XB + (1/140) * XC <= 40;
05: subject to B_limit: 0 <= XB <= 6000;
06: subject to C_limit: 0 <= XC <= 4000;
```

Code 6: Model PROD in AMPL

In Code 6 wird gezeigt, wie dieses einfache Modell mit der COM-Bibliothek formuliert werden kann, indem die Modellierungshilfskomponenten `MOVar` und `MORes` benutzt werden: Nach Dimensionierung wird zunächst in Zeile 4 die Zielfunktion festgelegt. Dazu muss der Methode `AddObjSum` eine alternierende Liste mit Koeffizienten und Variablen (als Name oder `MOVar`-Objekt) mitgegeben werden. Eine intuitivere Formulierung, wie z.B. die in C++ mögliche Form

```
M.Obj += coef1*Var1 + coef2*Var2 …
```

1 Anmerkung zu den Codebeispielen: Die nicht aufgeführte VB-Anweisung Imports MOPSComLibrary verhindert, dass allen Typen der COM-Bibliothek ein „MOComLibrary.“ vorangestellt werden muss. Kommentare beginnen mit einem Hochkomma.

ist mangels Operatorenüberladung[1] nicht möglich. `AddObjSum` fügt die noch nicht zum Modell `M` gehörenden Variablenobjekte `XB` und `XC` automatisch zu diesem hinzu. In Zeile 5 wird das Restriktionsobjekt `Time` hinzugefügt und Zeile 6 setzt die Nonzeros von `Time`, indem wieder eine alternierende Reihe von Koeffizienten und Variablen angegeben wird. Diese Syntax wurde gewählt, um unter den besagten Einschränkungen so näher an die algebraische Notation zu kommen. In den Zeilen 8 und 9 werden die Obergrenzen der Variablenobjekte gesetzt, wobei die Untergrenzen schon per Default auf 0 sind. Mit Optimize wird das Modell aus der `MOModel`-Komponente an den Solver übergeben, und nach erfolgreicher Optimierung werden die Ergebnisse in die `MOModel`-Komponente zurückübertragen.

```
01: Dim M As New MOModel
02: Dim XB, XC As New MOVar
03: Dim Time As New MORes

04: M.AddObjSum(25, XB, 30, XC)       ' 25*XB + 30*XC

05: M.AddRes(Time)
06: Time.Sum(1/200, XB, 1/140, XC)    ' (1/200)*XB + (1/140)*XC
07: Time.SetLT(40)                    ' <= 40

08: XB.UB = 6000                      ' 0 <= XB <= 6000
09: XC.UB = 4000                      ' 0 <= XC <= 4000

10: M.Optimize(OptDirection.Maximize)      ' Direction ist optional
```

Code 7: Model PROD mit Modellierungsobjekten

Das gleiche Modell wird in Code 8 mit matrixorientierter Modellierung, d.h. ohne Modellierungshilfsobjekte wie `MOVar`, `MORes` etc. generiert. Die matrixorientierte Modellgenerierung hat große Ähnlichkeit mit Benutzung der herkömmlichen MOPS.DLL, wobei die zentrale Komponente `MOModel` die Matrix darstellt. Die matrixorientierte Modellierung mit `MOModel` ist allerdings der DLL-Schnittstelle überlegen, da die Benutzung der Methoden z. B. durch optionale Argumente und Verwendung von `VARIANT`s bequemer und intuitiver ist. Darüber hinaus können auch mehrere Modelle gleichzeitig gehalten werden. Außerdem erlaubt `MOModel` dank integrierter Multithreading-Unterstützung (Methode `OptimizeAsync`) im Gegensatz zur MOPS.DLL Optimierungsläufe im Hintergrund.

Der Beispielcode ist eigentlich selbsterklärend, bis möglicherweise auf die Zeilen 5 und 6, die zwei Arten zeigen, ein Nonzero zu setzen: Einmal mit der Methode `AddNZ` und einmal mit der Eigenschaft `NZ`. Bei beiden können sowohl Indizes als auch Spalten- oder Zeilennamen angegeben werden.

---

[1] Im Gegensatz zur Version 2003 unterstützt VB.NET zwar ab der Version 2005 auch Operatorenüberladung, allerdings nicht in Verbindung mit COM-Objekten. Für C#, das schon immer Operatorüberladung kannte, gilt das gleiche.

```
01: Dim M As New MOModel

02: M.AddCol(0, 6000, MOColType.Continuous, 25, "XB")
03: M.AddCol(0, 4000, MOColType.Continuous, 30, "XC")

04: M.AddRow(-M.INF, 40, "Time")

05: M.AddNZ("Time", "XB", 1 / 200)     ' Setze NZ über Methode
06: M.NZ("Time", "XC") = 1 / 140       ' Setze NZ über Eigenschaft
```

Code 8: Model PROD mit Matrixgenerierung

Die Gegenüberstellung beider Beispiele sollte die unterschiedliche Sichtweise (objekt- vs. matrixorientiert) und einige Grundprinzipien der COM-Bibliothek zeigen, auch wenn hier noch keine wirklich überwältigenden Vorteile der Modellierungsobjektverwendung zu Tage treten. Im abschließenden Beispiel wird gezeigt, wie matrix- und objektorientierte Modellierung sogar vermischt werden können. Dort wird der bereits bestehenden Zeile „Time" das `MORes`-Modellierungsobjekt `Time` angehangen und darauf die `Sum`-Methode benutzt.

```
01: Dim M As New MOModel
02: Dim Time As New MORes

03: M.AddCol(0, 6000, MOComLibrary.MOColType.Continuous, 25, "XB")
04: M.AddCol(0, 4000, MOComLibrary.MOColType.Continuous, 30, "XC")

05: M.AddRow(-M.INF, 40, "Time")
06: Time.Attach(M, "Time")

07: Time.Sum(1/200, "XB", 1/140, "XC")
```

Code 9: Model PROD mit Matrixgenerierung plus Modellierungsobjekte

8.1.2 Erweitertes Produktionsmodell

Eine Verallgemeinerung des vorherigen, äußerst einfachen Produktionsmodells kann darin bestehen, eine beliebige Anzahl von Produkten zuzulassen. Code 10 zeigt das entsprechende AMPL-Modell aus [Fourer03, S. 8].

```
set P;
param a {j in P};
param b;
param c {j in P};
param u {j in P};
var X {j in P};

maximize Total Profit: sum {j in P} c[j] * X[j];
subject to Time: sum {j in P} (1/a[j]) * X[j] <= b;
subject to Limit {j in p}: 0 <= X[j] <= u[j];
```

Code 10: Model PROD2 mit AMPL

Das Beispiel in Code 11 ist ein Modellgenerator für das erweiterte Produktionsmodell, der unter anderem eine indizierte Variablenmenge (`MOVarSet`) als Modellierungsobjekt benutzt.

Das `MOVarSet X` wird in Zeile 10 durch die Initialisierung dem Modell `M` hinzugefügt. Es bekommt den Variablennamen „X“, und der einzelne Parameter `P` bedeutet, dass eine Initialisierung von 1 bis `P` stattfindet. Die Methode `Init` ist sehr flexibel benutzbar und kann neben solch einfachen Indizierungen auch komplexere und mehrdimensionale Indexierungen durchführen. In Zeile 12 werden die Obergrenzen für X aus dem Array `u` übernommen, und Zeile 13 setzt die Zielfunktionskoeffizienten, indem die Argumente der Methode `AddObjSum`, das Array `c` und das `MOVarSet X` als alternierende Folge von Koeffizienten und Variablen aufgefasst werden. Schließlich wird dem Modell in den Zeilen 14 bis 16 die Restriktion *Time* hinzugefügt.

Bemerkenswert ist die Kompaktheit des eigentlichen Modellgeneratorcodes, der kürzer ist als die vorangehenden Dimensionierungen und der vor allem völlig ohne Schleifen auskommt. Die Vermeidbarkeit von Schleifen bei vielen Modellierungsausdrücken ist ein wichtiges Merkmal der COM-Bibliothek, das darauf abzielt, in den Sprachen VBA und VB6, Schleifen vom langsamen Interpretercode in den schnellen C++-Code der COM-Komponenten zu verlagern.

```
01: Dim M As New MOModel          ' Modellinstanz
02: Dim X As New MOVarSet         ' Indizierte Variablenmenge
03: Dim Time As New MORes         ' Restriktionsobjekt
04: Dim P As Integer = 10         ' Anzahl Produkte
05: Dim b As Double = 1234        ' Kapazität
06: Dim a(P - 1) As Double        ' Menge pro Stunde von Produkt P
07: Dim ak(P - 1) As Double       ' Kehrwert von a()
08: Dim c(P - 1) As Double        ' Profit pro tonne von Produkt P
09: Dim u(P - 1) As Double        ' Maximalmenge pro Produkt P

10: 'Hier Arraydaten aus DB einlesen und berechnen …

11: X.Init(M, "X", P)    ' X.1, X.2, ... X.P
12: X.UB = u              ' X.1 bekommt UB aus u(0), X.2 aus u(1) usw.

13: M.AddObjSum(c, X)     ' c(0)*X.1 + c(1)*X.2 + ...

14: M.AddRes(Time)        ' Fügt M neues Restriktionsobjekt hinzu
15: Time.SetLT(b)         ' <= b
16: Time.Sum(ak, X)       ' Summe( ak(j-1)*X.j ) für j = 1 to P
```

Code 11: Modell PROD2 mit SafeArrays

Problematisch bei der Formulierung in Code 11 ist, dass P eine Zahl ist und nicht – wie im AMPL-Modell – eine Menge von Indexelementen. Dies liegt daran, dass VB-Arrays nur numerisch indizierbar sind, wobei die untere Indexgrenze in VB6/VBA frei wählbar ist und in VB.NET Null sein muss. Für solche Fälle sind die Komponenten `MOSet` und `MODataArray` vorgesehen, wie in Code 12 illustriert. Dort wird `P` als `MOSet` deklariert und in Zeile 10 mit drei Produktnamen initialisiert. Das Indexset `P` dient dann zur Initialisierung

der Arrays in den Zeilen 11 bis 14 und zur Indizierung des Variablensets in Zeile 16. Auf das erste Feld im `MODataArray a` würde beispielsweise zugegriffen mit `a.Value("ProdA")` oder kürzer mit `a("ProdA")`. Ansonsten gleicht Code 12 dem vorherigen Beispiel.

```
01: Dim M As New MOModel           ' Modellinstanz
02: Dim X As New MOVarSet          ' Indizierte Variablenmenge
03: Dim Time As New MORes          ' Restriktionsobjekt
04: Dim P As New MOSet             ' Produkte
05: Dim b As Double = 1234         ' Kapazität
06: Dim a As New MODataArray       ' Menge pro Stunde von Produkt P
07: Dim ak As New MODataArray      ' Kehrwert von a()
08: Dim c As New MODataArray       ' Profit pro tonne von Produkt P
09: Dim u As New MODataArray       ' Maximalmenge pro Produkt P

10: P.Add("ProdA", "ProdB", "ProdC")  ' Einlesen aus Array oder DB auch möglich
11: a.Init(P)
12: ak.Init(P)
13: c.Init(P)
14: u.Init(P)

15: 'Hier Arraydaten aus DB einlesen und berechnen …

16:  X.Init(M, "X", P)    ' X.ProdA, X.ProdB, X.ProdC
17:  X.UB = u             ' X.ProdA bekommt UB aus u("ProdA") usw.

18:  M.AddObjSum(c, X)    ' c("ProdA")*X.ProdA" + ...

19:  M.AddRes(Time)       ' Fügt M neues Restriktionsobjekt hinzu
20:  Time.SetLT(b)        ' <= b
21:  Time.Sum(ak, X)      ' Summe( ak(p)*X.p ) für alle p aus P
```

Code 12: Modell PROD2 mit MODataArrays

Bei den Beispielen Code 11 und Code 12 fällt auf, dass sich der Modellgenerator des Hilfsarrays `ak` für die Kehrwerte von `a` bedient, das zuvor gefüllt werden muss. Dies ist deshalb nötig, da in einer Programmiersprache wie VB ein Ausdruck wie `Time.Sum(1/a, X)` nicht möglich ist. Soll auf das Hilfsarray `ak` verzichtet werden, so kommt man in nicht umhin, Zeile 21 in Code 12 durch eine Schleifenkonstruktion zu ersetzen.

```
01-20: ' Wie in Code 12

21: Dim j As Object
22: For Each j In P                ' sum {j in P} (1/a[j]) * X[j]
23:   Time.Sum(1/a(j), X(j))
24: Next
```

Code 13: Modell PROD2 mit For...Each-Schleife

In der `For…Each`-Schleife wird über alle Elemente des Indexsets `P` iteriert, wobei die Laufvariable `j` unter .NET als Objekt deklariert wird, COM-intern aber ein `VARIANT` ist und somit sowohl Zahlen (Integer, Double) als auch Strings (BSTR) beinhalten kann. Hinter der simpel anmutenden `For…Each`-Schleife steckt intern eine komplexe Implementation des

Standardinterfaces `IEnumVARIANT`. Die Verwendung von `For…Each`-Schleifen über Aufzählungstypen ist häufig intuitiver, kompakter und dadurch oft auch weniger fehleranfällig als analoge Konstrukte mit numerischem Index (`For…Next`). Das soll auch das letzte Beispiel dieses Abschnittszeigen, in dem mit einer `For…Each`-Schleife die Lösungswerte der Variablen des Variablensets `X` ausgegeben werden. Auch hier verbergen sich hinter der augenscheinlich einfachen Benutzung im Hintergrund, d.h. auf COM-Ebene, komplexere Prozesse.

```
01-24: ' Wie in Code 13

25: M.Optimize(OptDirection.Maximize)

26: If M.SolverStatus = MOSolverStatusTyp.LPFinished And _
27:    M.SolutionStatus = MOSolutionStatusTyp.LPOptimalSolution Then
28:    ' Ausgabe Lösung
29:    Dim v As MOVar
30:    For Each v In X
31:        Debug.Print(v.Name & " LP-Sol= " & v.LPSol)
32:    Next
33: Else
34:   ' Fehleranalyse und -behandlung
35: End If

Beispielhafte Ausgabe:
X.ProdA LP-Sol= 123
X.ProdB LP-Sol= 456
X.ProdC LP-Sol= 789
```

Code 14: Modell PROD2 Lösungsausgabe

8.1.3 Diet-Modell

Das „Diet-Problem“ ist eines der in Lehrbüchern am häufigsten aufgeführten Beispiele für lineare Programmierung. Es geht auf Dantzig zurück (vgl. [Dantzig90]) und dient der Zusammenstellung eines kostenminimalen Diätplans unter bestimmten Nebenbedingungen bzgl. der notwendigen Nährstoffe. Die AMPL-Formulierung in Code 15 ist [Fourer03, S. 32] entnommen.

```
set NUTR;
set FOOD
param cost {FOOD} > 0;
param f_min {FOOD} >= 0;
param f_max {j in FOOD} >= f_min[j];
param n_min {NUTR} >= 0;
param n_max {i in NUTR} >= n_min[i];
param amt {NUTR,FOOD} >= 0;

var Buy {j in FOOD} >= f_min[j], <= f_max[j]
minimize Total_Cost: sum {j in FOOD} cost[j] * Buy[j]
subject to Diet {i in NUTR}:
n_min[i] <= sum {j in FOOD} amt[i,j] * Buy[j] <= n_max[i]
```

Code 15: Modell DIET in AMPL

Das folgende Codebeispiel Code 16 zeigt die Benutzung der Komponente `MOResSet`, die ein ein- oder mehrfach indiziertes Restriktionsset darstellt. In den Zeilen 16 bis 18 werden die Entscheidungsvariablen in Form des mit FOOD indizierten Variablensets `Buy` angelegt. In Zeile 20 wird dann das mit NUTR indizierte Restriktionsset `Diet` initialisiert. Anschließend werden dessen Ober- und Untergrenzen aus den `MODataArrays` `n_min` und `n_max` übernommen. Die Schleife in den Zeilen 23 bis 25 iteriert über die Elemente des Indexsets `NUTR`. Die Anweisung `Diet(i)` liefert ein einzelnes `MORes`-Objekt, also eine einzelne Restriktion aus dem Set. Auf diesem Objekt wird die `Sum`-Methode ausgeführt, die eine alternierende Folge von Koeffizienten und Variablen der Restriktion hinzufügt. Die Koeffizienten für diese Folge liefert der Ausdruck `amt(i)`, der die Zeile i aus dem zweidimensionalen `MODataArray` herausschneidet und diese intern als eindimensionales `SafeArray` zurückgibt. Die zugehörigen Variablen werden dem Variablenset `Buy` entnommen.

Es mag sein, dass diese implizite Formulierung nicht unmittelbar intuitiv erscheint, weshalb im Codebeispiel eine explizitere Alternative mit zwei verschachtelten Schleifen angegeben ist (Alternative A). Die die Argumente der `Sum`-Methode sind bei dieser Alternative ein einzelner Koeffizient als Resultat von `a(i,j)` und eine einzelne Variable als Resultat von `Buy(j)`.

Als weiteres Beispiel dafür, dass matrix- und modellierungsobjektorientierte Generierung beliebig gemischt werden können, dient die Alternative B in Code 16: Nachdem Variabeln und Restriktionen mithilfe der Modellierungsobjekte `MOVarSet` und `MOResSet` erzeugt wurden, werden nun die Nonzeros auf matrixorientierte Weise mit der Eigenschaft `NZ` des `MOModels` `M` gesetzt. Spalten- und Zeilennamen werden mit Stringoperationen erzeugt, die Verwendung absoluter numerische Spalten- und Zeilenindizes wäre aber auch erlaubt.

Kritisch bemerkt werden muss hier allerdings auch, dass aufgrund der syntaktischen Einschränkungen einer Programmiersprache keine der Alternativen die Kompaktheit und intuitive Verständlichkeit der AMPL-Formulierung erreicht.

```
01: Dim M As New MOModel
02: Dim NUTR, FOOD As New MOSet
03: Dim cost, f_min, f_max, n_min, n_max, amt As New MODataArray
04: Dim Buy As New MOVarSet
05: Dim Diet As New MOResSet
06: Dim i, j As Object

07: NUTR.Add("A", "B1", "B2", "C")      ' Vitamine
08: FOOD.Add("Beef", "Fish", "Ham")     ' Lebensmittel

09: cost.Init(FOOD)          ' Preis pro kg
10: f_min.Init(FOOD)         ' Mindestmengen Lebensmittel
11: f_max.Init(FOOD)         ' Höchstmengen Lebensmittel
```

```
12: n_min.Init(NUTR)         ' Mindestmengen Vitamine
13: n_max.Init(NUTR)         ' Höchstmengen Vitamine
14: amt.Init(NUTR, FOOD)     ' Vitaminmengen (Amounts) in jedem Lebensmittel

15: ' Daten einlesen z.B. aus Datenbank

16: Buy.Init(M, "Buy", FOOD)  ' Indiziertes Variablenset "Buy"
17: Buy.LB = f_min            ' Untergrenzen für das Variablenset
18: Buy.UB = f_max            ' Obergrenzen für das Variablenset

19: M.AddObjSum(cost, Buy)   Zielfunktion

20: Diet.Init(M, "Diet", NUTR)   ' Indiziertes Restriktionsset "Diet"
21: Diet.LB = n_min              ' LHSs
22: Diet.UB = n_max              ' RHSs
23: For Each i In NUTR
24:    Diet(i).Sum(amt(i), Buy)
25: Next

Alternative A für Zeilen 23-25 (explizitere Variante):
23: For Each i In NUTR
24:    For Each j In FOOD
25:       Diet(i).Sum(amt(i, j), Buy(j))
26:    Next
27: Next

Alternative B für Zeilen 23-25 (matrixorientiert):
23: For Each i In NUTR
24:    RowName = "Diet." & i                  ' Diet.A, Diet.B1 usw.
25:    For Each j In FOOD
26:       ColName = "Buy." & j                ' Buy.Beef, Buy.Fish usw.
27:       M.NZ(RowName, ColName) = amt(i,j)   ' Setzt Nonzero
28:    Next
29: Next
```

Code 16: Model DIET mit Modellierungskomponenten

8.1.4 Excel-Unterstützung

Die COM-Komponentenbibliothek beinhaltet auch Unterstützung für die Arbeit mit Excel, da die Benutzung aus VBA eines der Haupteinsatzfelder der Bibliothek darstellt. Das vorliegende Beispielmodell ist ein Problem aus der Personaleinsatzplanung, bei dem es darum geht, den Personalensatz in einem Unternehmen im Zeitrahmen von einer Woche zu minimieren, wobei zu beachten ist, dass eine bestimmte Mindestpersonalstärke an jedem Tag der Woche vorhanden sein muss und die Mitarbeiter Arbeitsverträge haben, die bestimmen, dass immer an fünf aufeinander folgenden Tagen gearbeitet wird und dann zwei Tage frei sind. Das Modell wird verschiedentlich in der Literatur aufgeführt, z.B. in [LINDO06, S. 194] und liegt auch als Beispiel dem Excel Add-In *ClipMOPS* bei.

	A	B	C	D	E	F	G	H	I	J
1	PERSONAL	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday	***TYP***	***RHS***
2	***Min***	1	1	1	1	1	1	1		
3	***LB***	0	0	0	0	0	0	0		
4	***UB***	105	105	105	105	105	105	105		
5	***TYP***	**INT**	**INT**	**INT**	**INT**	**INT**	**INT**	**INT**		
6	Monday	1			1	1	1	1	>=	17
7	Tuesday	1	1			1	1	1	>=	13
8	Wednesday	1	1	1			1	1	>=	15
9	Thursday	1	1	1	1			1	>=	19
10	Friday	1	1	1	1	1			>=	14
11	Saturday		1	1	1	1	1		>=	16
12	Sunday			1	1	1	1	1	>=	11
13										
14	***Activity***	2	6	0	6	0	4	5	**ObjFunc**	23

Abbildung 43: Excel-Beispiel Personaleinsatzplanung

Abbildung 43 zeigt das Tableau des Modells, wobei die Variablen *Monday*, *Tuesday* usw. die Anzahl der Mitarbeiter angeben, die ihren Fünf-Tages-Turnus am Montag, Dienstag usw. beginnen. Zeile 2 ist die Zielfunktion, die Zeilen 3 bis 4 sind die Unter- und Obergrenzen der Entscheidungsvariabeln und Spalte J gibt die pro Wochentag mindestens notwendige Anzahl von Mitarbeitern wieder.

Das so aufgebaute Tableau könnte mit dem Excel Add-In ClipMOPS gelöst werden, hier soll aber stattdessen die COM-Bibliothek benutzt werden, um deren flexible Interaktionsmöglichkeiten mit Excel zu demonstrieren (Code 17). Nach den Dimensionierungen werden in Zeile 4 die Tagesnamen in das Indexset `Days` übertragen, und anschließend wird in Zeile 6 das mit `Days` indizierte Variablenset `Personal` angelegt. Das Setzen der Zielfunktionskoeffizienten, sowie der Ober- und Untergrenzen benötigt jeweils nur eine einzige Anweisung. Die Bestimmung des Variablentyps in den Zeilen 10 bis 13 ist hingegen etwas umständlicher, da die einzelnen Excel-Zellen durchlaufen und geprüft werden müssen. `MOIndexSets` besitzen – wie andere Collection-Typen auch – eine `Count`-Eigenschaft und erlauben über die `Item`-Eigenschaft wahlfreien Zugriff auf einzelne Elemente, ähnlich einem Array.

In den Zeilen 14 bis 23 werden die Restriktionen aufgebaut. Hier wurde der Umweg über ein `MODataArray` genommen, um dessen Interaktionsmöglichkeiten mit Excel aufzuzeigen. Eine Alternative ohne `MODataArray` ist ebenfalls angegeben. Nachdem in Zeile 14 das mit `Days` indizierte Restriktionsset `Restr` angelegt wurde, wird der Inhalt der Excel-Spalten *TYP* und *RHS* in ein `MODataArray` kopiert. Die Zeilen dieses zweidimensionalen Arrays sind mit `Days` indiziert, und die beiden Spalten tragen die Namen *TYP* und *RHS*. Das `MODataArray` wird anschließend durchlaufen, wobei je nach Inhalt der Felder der Spalte *TYP* unterschiedliche Restriktionstypen gesetzt werden. Erklärungsbedürftig ist in diesem

Zusammenhang noch die Verwendung des `MOSets sHelp`, die beiden Spaltenindexwerte "TYP" und "RHS" als Argument seiner Default-Eigenschaft aufnimmt. Diese Default-Eigenschaft gibt eine Referenz auf sich selbst als Eigenschaftswert zurück. Dieser Trick des Interface-Designs ermöglicht eine sehr kompakte Syntax in vielen Fällen, in denen ein Indexset als Argument erwartet wird.

Das Setzen der Nonzeros in Zeile 24 ist sehr einfach: Die Eigenschaft `NZs` der `MOModel`-Komponente kann mit Excel-Ranges umgehen und liest die Matrix „en bloc“ aus der angegebenen Range. Nach Optimierung wird die optimale Lösung für die Personal-Variablen und der Zielfunktionswert in das Excel-Tabellenblatt geschrieben.

```
01: Dim M As New MOModel
02: Dim Days As New MOSet
03: Dim Personal As New MOVarSet
04: Dim Restr As New MOResSet

05: Days.Add Range("B1:H1")     ' Indexset

06: Personal.Init M, "Personal", Days  ' Variablenset Personal über Days indiziert
07: Personal.Cost = Range("B2:H2")     ' Zielfunktionskoeffizienten
08: Personal.LB = Range("B3:H3")
09: Personal.UB = Range("B4:H4")
10: For i = 1 To Days.Count                          ' Setzen des Variablentyps
11:    If Range("B5:H5").Item(1, i) = "INT" Then _    ' Continuous ist default
12:       Personal(Days.Item(i)).Type = MOComLibrary.Integer
13: Next

14: Restr.Init M, "Restr", Days         ' Restriktionsset indiziert über Days
15: Dim a As New MODataArray            ' Nimmt Typ- und RHS-Spalte temporär auf
16: Dim sHelp As New MOSet              ' Helper zur Indexseterzeugung
17: a.Init Days, sHelp("TYP", "RHS")    ' a wird 2-dimensionales Array
18: a = Range("I6:J12")                 ' und nimmt die Rage auf
19: For Each d In Days                  ' Enumersation über das Indexset
20:    If a(d, "TYP") = ">=" Then Restr(d).SetGT (a(d, "RHS"))  ' Setzte RHS
21:    If a(d, "TYP") = "<=" Then Restr(d).SetLT (a(d, "RHS"))
22:    If a(d, "TYP") = "="  Then Restr(d).SetEQ (a(d, "RHS"))
23: Next

24: M.NZs(Restr, Personal) = Range("B6:H12")  ' Setzen der Nonzeros

25: M.LocationOfMOPSDLL = "C:\MOPS\mops.dll"       ' Speicherort des Solvers
26: M.Optimize                                     ' Minimierung ist default

27: If M.SolutionStatus = IPOptimalSolution Then
28:    Range("B14:H14") = Personal.IPSol           ' Ausgabe Lösungswerte
29:    Range("J14") = M.IPObjFunctValue            ' Ausgabe opt. Zielfunktionswert
30: End If

31: M.WriteDebugModel "C:\temp\personal.debug.txt"

Alternative zu Zeilen 15-23:
For i = 1 To Days.Count
    If Range("I6:I12").Item(i, 1) = ">=" Then
        Restr(Days.Item(i)).SetGT (Range("J6:J12").Item(i, 1))
    Else
        If Range("I6:I12").Item(i, 1) = "<=" Then
            Restr(Days.Item(i)).SetLT (Range("J6:J12").Item(i, 1))
        Else
            Restr(Days.Item(i)).SetEQ (Range("J6:J12").Item(i, 1))
        End If
    End If
Next
```

Code 17: Modell PERSONAL in VBA

Sehr nützlich während der Modellgeneratorentwicklung ist die Möglichkeit, sich ein Debug-Modell (oder Teile davon) ausgeben zu lassen. Code 18 zeigt ein Beispiel.

```
! Debug model written by MOModel Component at 00:22:19 on 26.11.2007
MIN   Personal.Monday + Personal.Tuesday + Personal.Wednesday + Personal.Thursday +
      Personal.Friday + Personal.Saturday + Personal.Sunday

RESTRICTIONS
[1]Restr.Monday:  Personal.Monday + Personal.Thursday + Personal.Friday
                  + Personal.Saturday + Personal.Sunday  >= 17
[2]Restr.Tuesday:  Personal.Monday + Personal.Tuesday + Personal.Friday
                   + Personal.Saturday + Personal.Sunday  >= 13
[3]Restr.Wednesday:  Personal.Monday + Personal.Tuesday + Personal.Wednesday
                     + Personal.Saturday + Personal.Sunday  >= 15
[4]Restr.Thursday:  Personal.Monday + Personal.Tuesday + Personal.Wednesday
                    + Personal.Thursday + Personal.Sunday  >= 19
[5]Restr.Friday:  Personal.Monday + Personal.Tuesday + Personal.Wednesday
                  + Personal.Thursday + Personal.Friday  >= 14
[6]Restr.Saturday:  Personal.Tuesday + Personal.Wednesday + Personal.Thursday
                    + Personal.Friday + Personal.Saturday  >= 16
[7]Restr.Sunday:  Personal.Wednesday + Personal.Thursday + Personal.Friday
                  + Personal.Saturday + Personal.Sunday  >= 11

VARIABLE TYPES AND BOUNDS
[1]Personal.Monday:    Int   0   105
[2]Personal.Tuesday:   Int   0   105
[3]Personal.Wednesday: Int   0   105
[4]Personal.Thursday:  Int   0   105
[5]Personal.Friday:    Int   0   105
[6]Personal.Saturday:  Int   0   105
[7]Personal.Sunday:    Int   0   105
```

Code 18: Debug Model PERSONAL

8.2 Bewertung der Komponentenbibliothek

8.2.1 Bewertung nach allgemeinen Softwarequalitätsmerkmalen

Eine objektive Bewertung der entwickelten COM-Komponentenbibliothek anhand der in 5.3.1 aufgeführten allgemeinen Software-Qualitätsmerkmale nach ISO/IEC 9126 ist kaum möglich. Dies ist ein grundsätzliches Problem der Normierung von Softwarequalität, denn „im Gegensatz zu Gütern der Fertigungsindustrie bereitet bei der Softwareentwicklung die Operationalisierung dieser Qualitätsmerkmale […] trotz genormter Definitionen erhebliche Schwierigkeiten“ ([Herzwurm98, S. 59]). Auch die mit Modellen zur Qualitätssicherung befasste ISO-Norm 9000 Teil 3[1] resümiert: „Es gibt derzeit keine allgemein akzeptierten Meßmethoden für die Softwarequalität“. Vor diesem Hintergrund sei die folgende Anwendung der Software-Qualitätsmerkmale als Raster verstanden, innerhalb dessen lediglich tendenzielle Aussagen ohne die rigorose Objektivität von Maß- und Vergleichszahlen möglich sind.

Funktionalität

Die genannte Norm ISO 9126 gliedert *Funktionalität* in die Unterbegriffe *Richtigkeit, Ordnungsmäßigkeit, Interoperabilität, Sicherheit* und *Angemessenheit*.

[1] ISO 9000-3: 1992/27 (In dieser Fassung aber nicht mehr aktuell).

Richtigkeit und *Ordnungsmäßigkeit* des Funktionierens wurden im Entwicklungsprozess ständig überprüft und können zumindest aus Sicht des Entwicklers bejaht werden – eine endgültige Bewertung ist jedoch erfahrungsgemäß erst nach mehreren Praxiseinsätzen und Benutzerrückmeldungen möglich. Was die *Interoperabilität* anbelangt, so stellt die verwendete Basistechnologie COM eine größtmögliche Interoperabilität unter Windows sicher. Allerdings wird diese grundsätzliche Interoperabilität von der Tatsache eingeschränkt, dass die Modellierungsbibliothek explizit auf die Verwendung mit VB zugeschnitten wurde und in anderen Sprachen daher weit weniger einfach zu benutzen ist. Der Punkt *Sicherheit* wurde weitgehend ausgeklammert, da einerseits COM – anders als .NET – keine expliziten Sicherheitsmechanismen für Codeausführung, Benutzerrechte etc. kennt und andererseits hier auch kein besonderer Handlungsbedarf erkennbar ist. Es soll aber erwähnt werden, dass der gesamte C++-Code nur sichere Funktionen gem. den Visual C++ 2005 Spezifikationen benutzt, was die Gefahr von Sicherheitslücken durch Buffer-Overflows etc. erheblich mindert (z.B. Verwendung von `strcpy_s` anstatt `strcpy` etc.). Die Frage der *Angemessenheit* bedarf – mehr noch als die übrigen Begriffe – der genaueren Operationalisierung und lässt sich so leider kaum beantworten. Man kann höchstens ganz allgemein sagen, dass die Komponentenbibliothek ein probates, d.h. angemessenes Mittel, darstellt, einige der Probleme der Modellgeneratorenentwicklung zu vermindern.

Zuverlässigkeit

Der Begriff der *Zuverlässigkeit* wird in die Unterbegriffe *Reife*, *Fehlertoleranz* und *Wiederherstellbarkeit* aufgeteilt.

Da es sich bei der Komponentenbibliothek um eine prototypische Implementation handelt, mangelt es ihr in jedem Fall an *Reife*, was mögliche Auswirkungen auf die *Richtigkeit* (s. oben) der Ergebnisse haben kann. Auf den Bereich *Fehlertoleranz* wurde hingegen bereits per Design großen Wert gelegt: So werden fast alle Argumente von Methoden und Eigenschaften auf Validität überprüft, und im Fehlerfall wird fast immer eine explizite Fehlermeldung generiert. Dazu implementieren alle Komponenten das Interface `ISupportErrorInfo`. Auch in Bezug auf die Argumenttypen ist die Bibliothek sehr fehlertolerant, und insbesondere bei `VARIANT`-Argumenten wird versucht, einen falschen Typ durch Interpretation verwenden zu können. Erwartet beispielsweise eine Methode `MyMethod(VARIANT)` ein numerisches `VARIANT`-Argument, wird aber mit String-Argument aufgerufen (`MyMethod("1.23")`), so wird dies richtig interpretiert und ausgeführt. Das letzte Kriterium, *Wiederherstellbarkeit*, betrifft eher z.B. Datenbankanwendungen und lässt sich hier nicht sinnvoll anwenden.

Benutzbarkeit

Ziel der Entwicklung war, eine Bibliothek zu schaffen, die dem Benutzer in Bezug auf *Verständlichkeit*, *Erlernbarkeit* und *Bedienbarkeit* möglicht weist entgegenkommt, was bei einer Reihe von Designentscheidungen berücksichtigt wurde. So ist z.B. durch die häufige Verwendung von `VARIANTs` die Typstrenge stark herabgesetzt. Dies entspricht dem Programmiermodell von VB, und fördert Erlernbarkeit und Bedienbarkeit, auch wenn es sich für größere Projekte weniger eignet. Eine tatsächliche Bewertung der Benutzbarkeit wird aber erst möglich sein, wenn die Bibliothek, etwa im Rahmen der Lehre, Anwendung gefunden hat und Benutzerfeedback vorliegt.

Effizienz

Für genannte ISO-Norm besteht *Effizienz* aus *Zeitverhalten* und *Verbrauchsverhalten*. Im Gegensatz zu den zuvor aufgeführten Begriffen lassen sich diese beiden Punkte recht gut operationalisieren, bei einer gegenüberstellenden Bewertung läuft man allerdings oft Gefahr, Ungleiches miteinander zu vergleichen.

Vergleicht man etwa die Zeit, die benötigt wird, um iterativ 1.000.000 Variablen mit der MOPS.DLL (Funktion `PutCol`) und der COM-Komponente `MOModel` (Methode `AddCol`) anzulegen, so ist die Komponente ungefähr um dem Faktor 4 langsamer. Dies hat drei Gründe: Erstens benutzt die Komponente ein völlig dynamisches Speichermanagement, d.h. im Gegensatz zur MOPS.DLL, die in der Funktion `AllocateMemory` den benötigten Speicher statisch im Voraus anlegt, geschieht dies in den COM-Komponenten jedes Mal, wenn intern ein neues Objekt instanziert und den Vector-Containern hinzugefügt wird – was entsprechend Zeit kostet. Zweitens ist die interne Modellhaltung der MOPS.DLL um Einiges schlanker als die der `MOModel`-Komponente. Letztere benutzt intern die objektorientierten Datentypen der STL, die zwar dynamische Speicherverwaltung erleichtern, aber auch einen gewissen Overhead implizieren. Drittens hängt die schlechtere Performance der Komponente auch mit der ausgiebigen Verwendung von `VARIANTs` zusammen, was sich u.a. im Fall der `AddCol`-Methode zeigt: Jeder im Aufruf übergebene `VARIANT` muss intern eine explizite Typprüfung und evtl. Typumwandlung durchmachen, was Rechenzeit kostet.

Auch wenn bei solchen Gegenüberstellungen des Zeitverhaltens die Andersartigkeit der beiden Konzepte berücksichtigt werden muss, so soll dennoch nicht verschwiegen werden, dass derzeit die Laufzeit-Performance der COM-Komponenten dem Vergleich mit anderen Systemen noch nicht standhält. Dies liegt einerseits an unvermeidbaren Overheads, wie den ge-

nannten Typchecks und Typumwandlungen von `VARIANTs`, andererseits aber auch an zunächst aus Zeitgründen in Kauf genommenen „Suboptimalitäten“, was das interne Design einiger Routinen und Datenstrukturen anbelangt. Primäres Designziel war nämlich nicht der Entwurf einer möglichst performanten Komponentenbibliothek für große Modelle, sondern die Entwicklung einer einfach zu benutzenden Bibliothek für die Ad-hoc-Generierung kleinerer und mittlerer Modelle unter VB. Viele der derzeit noch bestehenden Performanceengpässe lassen sich durch eine Überarbeitung des Codes lösen, sind also eher eine Frage der zu investierenden Entwicklungszeit.

Die zweite Komponente des ISO-Effizienzbegriffs, das *Verbrauchsverhalten*, kann hingegen weit günstiger bewertet werden: Verbrauchsverhalten bedeutet hier vor allem Speicherverbrauchsverhalten, das wegen der bereits erwähnten dynamischen Speicherverwaltung sehr gut mit der Modellgröße skaliert und wenig Speicher-Overhead benötigt.

Änderbarkeit

Der Begriff der *Änderbarkeit* spaltet sich auf in *Analysierbarkeit*, *Modifizierbarkeit*, *Stabilität* und *Prüfbarkeit*.

Dank ausgiebiger Kommentierung ist die Komponentenbibliothek gut *analysierbar*. Ihre durchgängige Objektorientierung wirkt nicht nur positiv auf die *Stabilität*, sondern verringert auch die Gefahr von Seiteneffekten und macht sie so *modifizierbar*. Der *Prüfbarkeit* der Modellgenerierung dient u.a. die Ausgabemöglichkeit von Debug-Modellen.

Übertragbarkeit

Anpassbarkeit, *Installierbarkeit*, *Konformität* und *Austauschbarkeit* sind in der ISO-Norm die Unterbegriffe der *Übertragbarkeit*.

Die *Installierbarkeit* der Bibliothek ist einfach und geschieht nach COM-Standard durch Registrierung der Komponenten in der Registry des Rechners. Das gilt auch im Fall einer Distribution im Rahmen einer Fremdanwendung. *Anpassbarkeit* durch Codemodifikation ist jederzeit möglich, allerdings ist zu beachten, dass Veränderungen der Interfaces nach Publikation unterbleiben müssen, um die Integrität bestehender Anwendungen zu schützen. Nach COM-Standard muss in diesem Fall ein komplett neues Interface erstellt werden. Zu *Konformität* und *Austauschbarkeit* kann keine Aussage gemacht werden, da es in diesem Bereich keine Normen gibt, zu denen man Konformität herstellen könnte und ein Austausch ebenfalls keinen Sinn macht.

8.2.2 Bewertung nach Erfüllung spezifischer Anforderungen

Dieser Abschnitt soll kurz darauf eingehen, inwiefern die COM-Bibliothek die Anforderungen an Programmierschnittstellen von Solversystemen nach Abschnitt 5.3.2 erfüllt.

- **Zielkontext-Adäquanz:** Die Bibliothek stellt als Teil des Gesamtschnittstellenkonzepts eine zielkontextadäquate Ausprägung für den VB/VBA/VB.NET-Kontext dar, in den sie sich bezüglich Syntax und Benutzungsweise gut einfügt (siehe Beispiele in 8.1). Außerhalb dieses Zielkontextes ist die Benutzung aber umständlich, und es sollten dort andere Module des Gesamtschnittstellenkonzepts benutzt werden, wie z.B. die C++-Modellierungsbibliothek für einen C/C++-Kontext.
- **Modellierungssprachenunterstützung:** Ist mit den Sprachen AMPL, MathProg, Lindo und MPL sehr gut erfüllt. Im Fall von AMPL ist mit der Methode `ExecuteAMPL` sogar Datenaustausch von Solver und Modellierungssprache in beide Richtungen möglich (vgl. Anforderungen in [Schichl04, S. 51]).
- **Daten- und Solverschnittstellen:** An Datenschnittstellen besitzt die vorliegende Implementation nur eine Excel-Schnittstelle. Weitere Interfaces zu Datenbanken, XML etc. sind konzeptionell vorgesehen, aber noch nicht implementiert. Gleiches gilt für die Anbindung weiterer Solver.
- **Performanz:** Hier existieren, wie bereits im vorherigen Abschnitt dargelegt, noch größere Schwachstellen, die sich aber durch weiteren Implementationsaufwand verringern lassen.
- **Rapid Model Development:** Mit der Komponentenbibliothek unter VB sehr gut möglich, allerdings weniger schnell als bei Benutzung einer Modellierungssprache
- **Gleichzeitige Verwendung mehrerer Modelle:** Wird unterstützt
- **Multithreading:** Wird unterstützt mit Methode `OptimizeAsync`.
- **Lauffähigkeit unter 64-Bit:** Noch nicht implementiert, aber problemlos möglich
- **Callbacks:** Die neuen Callbacks der MOPS.DLL werden benutzertransparent von der den Modellierungskomponenten verwendet
- **Vielfältige Modellformate:** Unterstützung für MPS, Triplet, Lindo ist bereits implementiert, weitere Modellformate sind eingeplant
- **Debugging-Unterstützung:** Ausgabe von Debug-Modellen und Log-Files, sowie Exceptions mit expliziten Meldungen

Insgesamt kann also eine zwar nicht vollständige, aber doch weitgehende Abdeckung der Anforderungen festgestellt werden, auch wenn einige Features noch nicht vollständig implementiert sind.

8.3 Bewertung von MOPS Studio

Auch MOPS Studio, als zweiter Implementationsschwerpunkt dieser Arbeit, soll hier nach allgemeinen Softwarequalitätskriterien (ISO 9126, Abschnitt 5.3.1) und nach Abdeckungsgrad spezifischer Anforderungen an Endbenutzerschnittstellen (Abschnitt 5.3.3) bewertet werden, dies allerdings in etwas knapperer Form, um zusätzlich noch einige Erfahrungen aus dem Praxiseinsatz des Programms wiederzugeben.

8.3.1 Bewertung nach allgemeinen Softwarequalitätsmerkmalen

Funktionalität

Was die Unterpunkte der Funktionalität *Richtigkeit* und *Ordnungsmäßigkeit* anbelangt, so sind keine Fehler bekannt, die falsche oder nicht ordnungsgemäße Ergebnisse erzeugten. Die Interoperabilität mit anderen Solvern oder Modellierungssystemen ist durch die Unterstützung unterschiedlicher Modellformate (MPS, AMPL etc.) gewährleistet, könnte aber noch auf weitere Formate ausgedehnt werden. Dem Unterpunkt *Sicherheit* wurde keine expliziten Programmfunktionen gewidmet, es gibt also keine Benutzerverwaltung oder Ähnliches – was allerdings auch kaum Sinn machen würde. Als .NET-Programm unterliegt MOPS Studio aber den standardmäßigen Sicherheitsfeatures dieses Frameworks.

Zuverlässigkeit

In Punkto *Reife* als Unterbegriff der Zuverlässigkeit kann gesagt werden, dass MOPS Studio einen Reifegrad erreicht hat, der kurz vor dem Übergang zur kommerziellen Verwertbarkeit steht. MOPS Studio ist dank der frühen Erprobung um Universitätsumfeld und einer Reihe von iterativen Verbesserungen und Korrekturen erheblich reifer als die COM-Komponentenbibliothek. Auch die *Fehlertoleranz* ist sehr hoch, da fast alle Benutzereingaben verifiziert werden. Einen Programmabsturz durch Fehleingaben zu provozieren ist daher kaum möglich.

Benutzbarkeit

Die Benutzbarkeit kann als sehr hoch eingestuft werden. Die Oberfläche ist fast selbsterklärend und bedarf nicht notwendigerweise einer Dokumentation (die in den ersten Versionen auch fehlte), was kaum von Benutzerseite bemängelt wurde.

Effizienz

In Bezug auf das *Zeitverhalten* kann man feststellen, dass dieses mittlerweile auch für größere Modelle an kommerzielle Systeme heranreicht. Damit ist allerdings nicht die Lösungszeit der Optimierung gemeint, die Sache des Solvers ist, sondern das Laden und Speichern von Projekten, Formatkonvertierungen, die GUI-Performance etc. Allerdings sind z.B. bei der Geschwindigkeit des Parsings von MPS-Modellen durch die `MOModel`-Komponente und beim Schreiben von Solution-Summary-Files noch Verbesserungen möglich. In Bezug auf das *Verbrauchsverhalten*, als zweiter Komponente des ISO-Effizienzbegriffs, muss angemerkt werden, dass hier bei sehr großen Modellen Engpässe eintreten, die aber in der Regel nicht an MOPS Studio, sondern am Speicherverbrauch der Modellierungssprachen (insbesondere MathProg) oder generell an der 2 GB-Grenze des Prozessraums unter 32-Bit-Betriebssystemen liegen.

Änderbarkeit

Durch reichhaltige Kommentierung und modulares Design weist MOPS Studio eine gute *Analysierbarkeit* und *Modifizierbarkeit* auf. Auch die *Stabilität* ist nach bisherigen Erfahrungen sehr hoch, und (Nach-)*Prüfbarkeit* der Ergebnisse kann durch Gegenrechnungen mit anderen Systemen (z.B. MPL Studio, AMPLWin etc.) erfolgen.

Übertragbarkeit

Der letzte der ISO-Softwarequalitätsbegriffe setzt sich aus *Anpassbarkeit*, *Installierbarkeit*, *Konformität* und *Austauschbarkeit* zusammen. Hier soll davon nur die einfache Installierbarkeit hervorgehoben werden, die auf der Basis von InstallShield erfolgt und im Laufe mehrerer Versionen immer weiter verbessert wurde.

8.3.2 Bewertung nach Erfüllung spezifischer Anforderungen

Die spezifischen Anforderungen an Endbenutzerschnittstellen aus 5.3.3 sind wie folgt erfüllt:

- Grafische mathematische Notation: Nein, nur textuell
- Ansichten der Modellstruktur mit unterschiedlicher Granularität: Nein
- Syntax-orientiertes Editing: Syntax coloring ja, aber kein Syntaxcheck in Echtzeit
- Interaktive Hilfe: Ja, Online Hilfe, Samples und Tutorial
- Visualisierung der Zusammenhänge zwischen Modellteilen: Nein
- Visualisierung des Lösungsprozesses: Textuell ja, aber nicht grafisch

- Visualisierung der Ergebnisse: Textuell ja, nicht grafisch
- Offene Modellaustauschformate: Ja (MPS u.a.)
- Projekte zur Zusammenfassung von Modellen, Daten und Einstellungen: Ja
- Modelexplorer: Nein
- Dataexplorer: Nein (in Entwicklung)
- Page templates, GUI-Constructor (ähnlich AIMMS): Nein
- Case- und Szenariomanager für Modellvarianten: Nein
- Anbindung verschiedener Solver: Prinzipiell ja, derzeit ist nur MOPS angebunden
- Verschiedene Modellierungssprachen: Ja
- Unterstützung für verteilte Optimierung: In Entwicklung
- Intelligente Fehlermeldungen mit Korrekturvorschlägen: Ja, meist ohne Korrektur
- Debugging: Teilweise. (Ausgabe von Debug-Modellen, aber kein Stepping durch prozedurale Modellteile)
- Grafischer Matrix-Explorer mit Zoom- und Scroll-Funktionen: Nein (in Entwicklung)
- Erweiterte Visualisierungsmöglichkeiten für Ergebnisse: Nein

Zusammenfassend kann man sagen, dass MOPS Studio bereits eine Reihe grundlegender Anforderungen an ein integriertes Modellierungssystem erfüllt, wobei einige Features auf dem Weg zu einem kommerziellen Produkt noch weiterentwickelt werden müssen.

8.3.3 Bewertung des Einsatzes in der Lehre

MOPS Studio wurde seit 2005 an den Universitäten Paderborn, FU-Berlin und Köln im Rahmen der Lehre in OR-Übungen eingesetzt. Hierbei wurde vor allem mit AMPL- und MathProg-Modellen gearbeitet. Die Studenten wurden von den Dozenten gebeten, eventuelle Fehler an die Entwickler von MOPS und MOPS Studio zu melden. An der Uni Paderborn stand der Autor dieser Arbeit auch in den Foren des studienbegleitenden Online-Systems (OpenSMT, opensmt.org) für Feedback und Hilfestellung zur Verfügung. Außerdem gibt es in MOPS Studio eine Funktion *Report Bug*, die explizit zur Meldung von Fehlern einlädt (Abbildung 44).

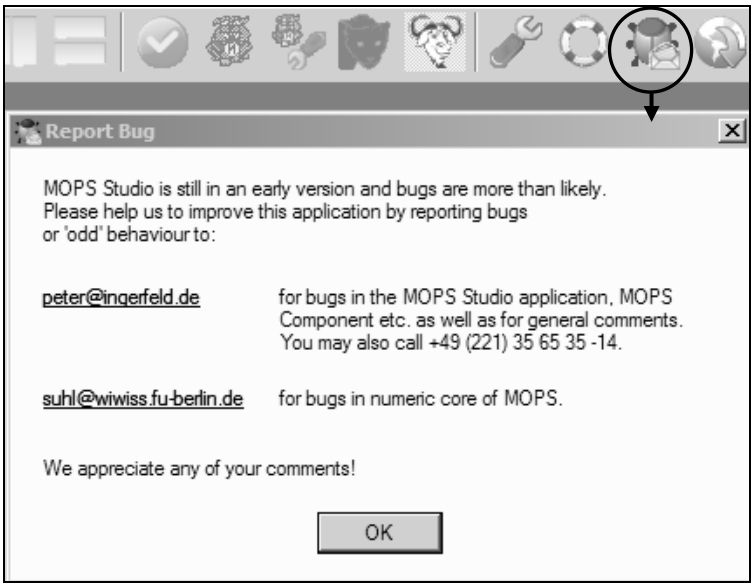

Abbildung 44: Report Bug

Sehr positiv für die Qualität des Produkts MOPS Studio war dabei die Tatsache, dass es insgesamt nur zu sehr wenigen Fehler-Rückmeldungen kam (schätzungsweise unter 15). Fast alle Fragen und Probleme betrafen Installationsschwierigkeiten, denn die ersten Versionen von MOPS Studio (bis v1.3) waren für das .NET Framework 1.1 geschrieben. Oft fehlte dieses auf den Installationsrechnern, oder die Benutzer hatten bereits Version 2.0 (aber nicht Version 1.1) installiert. Zwar wird das Vorhandensein des richtigen Frameworks vom Installer geprüft, aber offenbar war einigen Benutzern nicht klar, dass eine parallele Installation beider Frameworkversionen auf dem gleichen System problemlos möglich ist. Einige Rückmeldungen bezogen sich auch auf Fehlermeldungen, die nach Lizenzablauf auftauchen (In der Studentenversion von MOPS Studio ist die MOPS.DLL größenlimitiert und zeitlich begrenzt). Von allen Fehler-Rückmeldungen betraf allerdings nur eine Einzige einen tatsächlichen Fehler. Dieser Bug im MPS-Writer wurde umgehend behoben.

Einige wenige Feedback-Meldungen enthielten Verbesserungsvorschläge für die Benutzbarkeit, die bereits alle in der derzeitigen Version umgesetzt worden sind.

Insgesamt kann man MOPS Studio eine sehr geringe Fehlerquote und eine hohe Reife an der Schwelle zur kommerziellen Verwertbarkeit zuschreiben.