6 Konzeption einer Middleware für Optimierungssoftware

Im Folgenden soll unter Berücksichtigung der zuvor aufgeführten Problembereiche und Anforderungen ein universelles Schnittstellensystem zur Einbettung mathematischer LP-/MIP-Optimierungssoftware in Anwendungssysteme konzipiert werden. Da die Implementation exemplarisch mit dem Optimierer MOPS durchgeführt wurde, steht dieser Solver bei manchen Details im Vordergrund. Das Konzept bezieht sich aber ausdrücklich auch auf andere Solver und soll als generalisiertes Optimizer-Middlewarekonzept verstanden werden.

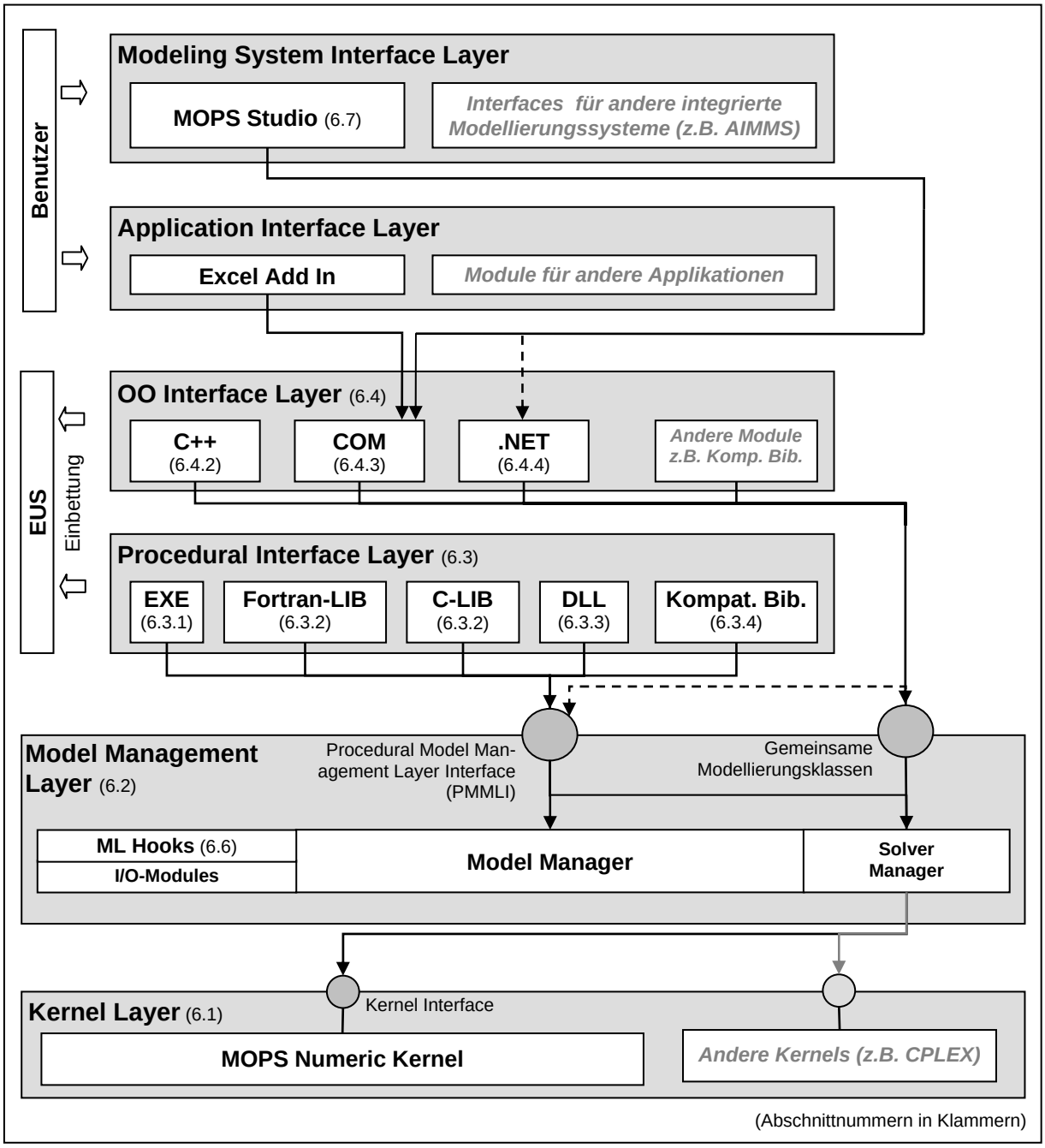

Abbildung 23: Konzept Schnittstellensystem

Grundgedanke ist die Modularisierung, die schichtweise Anordnung und die Loslösung des Schnittstellensystems vom numerischen Kern. Ziel ist die Zurverfügungstellung von kontext-adäquaten Schnittstellenmodulen, die baukastenartige Erweiterbarkeit um zusätzliche Module sowie die Austauschbarkeit des numerischen Kerns.
Abbildung 23 skizziert die einzelnen Schichten des Konzepts und ihre Bestandteile:
Die unterste Ebene wird vom *Kernel Layer* gebildet, der den numerischen Kern von MOPS oder eines anderen Solvers und damit die gesamte numerische Optimierungslogik enthält. Von den darüber gelegenen Schichten wird der Numeric Kernel als Black Box betrachtet und ausschließlich über das *Kernel Interface* angesprochen.
Die komplette Modellverwaltung ist auf dem *Model Management Layer* im *Model Manager* angesiedelt. Dieses Modul übernimmt für die darüber gelegenen Schichten und Schnittstellen alle zentralisierbaren Aufgaben der Modellverwaltung, der Ein- und Ausgabe und der Verifikation. Der Model Manager speichert und verwaltet ein oder mehrere Modelle und übergibt sie zur Optimierung an den Solver Manager. Ebenfalls zum Model Management Layer gehören die Schnittstellen zu Modellierungssprachen (*Modeling Language Hooks*), die der Model-Manager zur Anbindung unterschiedlicher Modellierungssprachen nutzt. Alle zentralisierbaren Input- und Output-Aufgaben werden ebenfalls auf dem Model Management Layer ausgeführt (*IO-Modules*). Hierzu zählen vor allem Schreib- und Leseroutinen für eine möglichst große Zahl unterschiedlicher Modellformate (fixed/free MPS, Triplet, XML basierte Modellformate, Lindo usw.), aber auch alle Reportingfunktionen wie die Ausgabe von Statistik- und Lösungsfiles. Der *Solver Manager* ist ein Modul, das alle solverspezifischen Funktionalitäten kapselt. Es nimmt vom Model Manager das Modell entgegen und übergibt es an das Kernel Interface des jeweiligen Solvers. Der Solver Manager verwaltet alle solverspezifischen Parameter. Solange nur MOPS angebunden wird, gibt es auch nur einen Solver-Manager, weitere Kernels haben jeweils einen eigenen Solver-Manager.
Oberhalb des Model Management Layers, auf jeweils höheren Aggregations- und Abstraktionsniveaus liegen die Programmier- und Benutzerschnittstellen. Die Programmierschnittstellen teilen sich in einen objektorientierten und einen prozeduralen Layer. Die Benutzerschnittstellen umfassen den *Modeling System Interface Layer* und den *Application Interface Layer*. Die beiden Programmierschnittstellen-Layer enthalten kontext-adäquate Interfaces in modularisierter Form für eine Reihe von Programmierumgebungen bzw. Frameworks. Die einzelnen Interfacemodule fügen sich bezüglich Syntax und sonstigen Features weitestmöglich in die jeweilige Zielumgebung ein. Das modulare Konzept ist dabei offen für weitere Programmierschnittstellen. Zusätzlich zu den in Abbildung 23 dargestellten Interfaces wären z.B. noch

spezifische Interfaces für Java und andere Sprachen denkbar. Ebenfalls sind so genannte „Kompatibilitätsbibliotheken“ vorgesehen, die das Programmierinterface eines anderen Optimierers, ganz oder teilweise emulieren und so die Austauschbarkeit des Solvers in bestehendem Code ermöglichen. Eine solche Austauschbarkeit via Kompatibilitätsbibliotheken hat insbesondere einen marktbezogenen Hintergrund: Durch die Emulation von Teilen der Solverinterfaces kommerzieller Solver, wie z.B. CPLEX, wird einem weniger marktpräsenten Solver (wie etwa MOPS) der Weg in bestehende Anwendungen geebnet, ohne dass der Verwender bzw. Kunde das Investitionsrisiko der Neuanpassung seiner Anwendung einginge.
Der *Application Interface Layer* umfasst Plug-In-Schnittstellen für Standardanwendungen, die den Benutzer dieser Anwendungen direkt und indirekt mit dem Optimierer arbeiten lassen. Das mit Abstand wichtigste Modul auf dem Application Interface Layer ist dabei die Excel Add-In-Schnittstelle. Zwar verfügt Excel bereits über einen eingebauten Solver (vgl. [Fylstra98]), für anspruchsvolle LP-/IP-Modelle ist dieser jedoch Produkten wie Clip-MOPS weit unterlegen. Angesichts des weiten Feldes von Problemstellungen, die mit Spreadsheet-Solvern lösbar sind (vgl. [Ragsdale04]), ist die Spreadsheet-Anbindung ein wichtiges Feature. Weitere Plug-Ins für andere Anwendungen (z.B. aus den Bereichen Mathematik, Finanz- oder Ingenieurwesen) sind ebenfalls denkbar, dürften aber im Vergleich zur Spreadsheet-Anbindung weit weniger relevant sein.
Der *Modeling System Interface Layer* umfasst die Schnittstellen zu integrierten Modellierungssystemen. Hier ist zunächst das im Rahmen dieser Arbeit entwickelte MOPS Studio zu nennen, darüber hinaus wären aber auch Schnittstellenmodule für andere, solver-offene Modellierungssysteme, wie z.B. AIMMS, möglich.
Im Folgenden werden nun die einzelnen Bestandteile des Konzepts näher erläutert.

6.1 Kernel Layer

Der Kernel Layer enthält in modularer Form die numerischen Kernels der Solver. Hier wird zunächst nur ein Kernelmodul für MOPS betrachtet, das Konzept ist aber ausdrücklich offen für weitere Solver, die über ein jeweils eigenes Kernel Interface angesprochen werden. Das Kernel Interface weist den geringstmöglichen Funktions- und Komplexitätsumfang auf, was die Austauschbarkeit des Kernels fördert und ihn von nahezu allen schnittstellenbezogenen Aufgaben befreit. Das Kernel Interface erhält aus dem darüber liegenden *Solver Manager* ein korrektes Modell, so dass im Kernel Layer keine Überprüfung der formalen Validität mehr vorgenommen werden muss. Ein Kernel Interface für MOPS der 2007 aktuellen Version 8.x bräuchte lediglich folgende Funktionalitäten zu unterstützen:

- Übergabe eines Modells (`PutModel`)
- Übergabe von Sets (`PutSOS`)
- Übergabe und Abfrage von Parametern (`SetParameter, GetParameter`)
- Start und Fortsetzen der Optimierung (`Optimize`)
- Abfrage Optimierungsstatus (`GetParameter`)
- Rückgabe der Optimierungsergebnisse (`GetSolution`)

Für die ab MOPS Version 9.x (2008) hinzugekommen Callbacks oder falls andere callbackfähige Kernels eingebunden werden, müsste man das Kernel Interface zumindest um eine Funktion zum Setzen von Callbackpointern erweitern. Werden weitere Solver eingebunden, besitzen diese leicht abgewandelte oder ergänzte eigene Kernel Interfaces. Ergänzungen sind z.B. auch erforderlich, wenn der Solver benutzergesteuerte Cuts und Branchingregeln erlaubt oder die Analyse von Ranges, IIS oder IUS unterstützt. Für einen einzubindenden Fremdsolver besteht das Kernel Interface aus einem Subset dessen normaler Schnittstellen mit geringstmöglichem Funktions- und Komplexitätsumfang.

6.2 Model Management Layer

Der Model Management Layer besteht aus dem *Model Manager*, mit seinen I/O-Modulen und Modellierungssprachenschnittstellen sowie dem *Solver Manager*. Als Schnittstellen besitzt der Layer das *Procedural Model Management Layer Interface (PMMLI)* und die *Gemeinsamen Modellierungsklassen*.

Der **Model Manager** ist der Kern der zentralisierten Modellverwaltung und übernimmt für die darüber liegenden Schichten alle Aufgaben, die nicht kontextschnittstellenspezifisch sind und die in wenigstens zwei Modulen in gleicher Form anfallen. Er kann mehrere Modelle gleichzeitig verwalten und speichert diese in einem für die Interaktion mit Programmier- und Benutzerschnittstellen optimierten internen Format ab. Auf den Datenstrukturen dieses internen Modellformats müssen vor allem folgende Operationen schnell und ressourceneffizient durchgeführt werden können:

- Hinzufügen von Variabeln, Restriktionen und Nonzeros
- Einfügen von Variabeln und Restriktionen an beliebigen Positionen
- Löschen von Variablen, Restriktionen und Nonzeros
- Schnelles Suchen und Einfügen von Variablen- und Restriktionsnamen
- Umwandlung in das vom Kernel Interface benötigte Format

Der Modell Manager muss dabei mit einer dynamischen und skalierbaren Speicherverwaltung arbeiten, da die Größe der Modelle stark variieren kann und der Speicherbedarf vorab nicht bekannt ist. Das Konzept sieht die Verwaltung mehrer Modelle, nicht jedoch die mehrerer Environments vor. Environments, wie sie etwa CPLEX oder LINDO kennen, haben zwar Vorteile (siehe 3.1.2.1), blähen aber Funktionsaufrufe mit einem zusätzlichen Parameter auf. Wenn ein Schnittstellensystem Funktionen zum Kopieren von Parametern und zum Setzen von Parametern und Einstellungen für ganze Modellgruppen besitzt, kann auf explizite Environments verzichtet werden.

Der **Solver Manager** kapselt alle solverspezifischen Aufgaben, wie z.B. die Verwaltung der Solver-Parameter, die Bedienung der Kernel Interfaces und das Management paralleler Optimierungsläufe.

Alle Zugriffe auf den Model-Manager und den Solver-Manager laufen über das *PMML-Interface,* oder über die *Gemeinsamen Modellierungsklassen* ab, die beide im Vergleich zum schlanken Kernel Interface einen erheblich größeren Funktionsumfang bieten.

Das **Procedural Model Management Layer Interface (PMMLI)** wird primär von den Schnittstellenmodulen des Procedural Interface Layers benutzt, kann aber auch von anderen darüber liegenden Schichten angesprochen werden. Die Funktionen des PMMLI sind in Anhang 6 aufgelistet. Sie wurden unter dem Gesichtspunkt der Zentralisierung aus den gemeinsamen Funktionen prozeduraler Solverinterfaces (Anhang 1) abgeleitet und sind nach dem dort aufgeführten Schema gegliedert. Vor dem Hintergrund der exemplarischen Implementation für MOPS fallen einige der in 3.1 aufgeführten Funktionalitäten anderer Solver z.B. aus den Bereichen Cuts, Lösungsprozessteuerung oder Infeasibility-Analyse weg. Bei einer möglichen Erweiterung und Einbeziehung anderer Solverkernels müssten diese Bereiche ggf. im Model-Manager und in darüber liegenden Programmierschnittstellen nachgetragen werden. Solche Erweiterungen berauben das Konzept jedoch eines Teils seiner Universalität für möglichst viele Solver, da sie eine Entfernung vom Prinzip des kleinsten gemeinsamen Nenners bedeuten. Das Model Manager Interfaces ist jederzeit erweiterbar oder abänderbar, da es nicht veröffentlicht ist und nur zur internen Benutzung dient.

Die **Gemeinsamen Modellierungsklassen** dienen den Modulen des OO Interface Layers zur Wiederverwertung und als Interface (siehe Abschnitt 6.4.1).

Schließlich seien noch die folgenden Anmerkungen zu **Implementationsaspekten** des Model Management Layers gemacht:

Als geeigneter Bibliothekstyp bietet sich eine statische Linklibrary (LIB) an, die zu den Modulen des Procedural Interface Layers und denen des OO Interface Layers hinzugelinkt werden kann. Denn während der Numeric Kernel aus Gründen der Austauschbarkeit und seinem schlanken, sehr invarianten Interface sinnvollerweise als DLL implementiert wird, muss bei den Interfaces des Model Management Layers in Betracht gezogen werden, dass deren Funktionsumfang erheblich größer ist und dass die Wahrscheinlichkeit von nachträglichen Veränderungen (Versionen) hoch ist. Um die bekannten Probleme der „DLL Hell“ (vgl. [Szyperski99]) zu vermeiden, sollte der Layer daher als statische Library implementiert werden, die die Funktionen des PMMLI und die Gemeinsamen Modellierungsklassen exportiert.
Um in allen Umgebungen unter Windows lauffähig zu sein, muss nativer, unmanaged Code verwendet werden – zumal der numerische Kernel von MOPS und anderer Systeme ohnehin aus nativem Code besteht, so dass in jedem Fall, auch bei der Integration in eine .NET-Anwendung, eine gemischte managed/unmanaged Codebasis zum Laufen kommt. Als Basissprache für die Implementation aller Module kommt nur C++ in Frage, da diese Sprache die flexibelste und systemnächste Programmierung erlaubt. Dies gilt auch für das .NET-Interface, da C++ (Visual C++ 2005) als einzige Sprache ein fast beliebiges Mischen von managed und unmanaged Code erlaubt.
Die vom Procedural Model Manager Interface und den Gemeinsamen Modellierungsklassen verwendeten Datentypen müssen der kleinste gemeinsame Nenner für alle zugreifenden Module sein. Spezifische Datentypen eines Frameworks oder einer Programmiersprache (z.B. BSTR, SafeArrays, UnsignedLong etc.) scheiden damit aus, so dass eine Beschränkung auf folgende Typen opportun ist:

- Long (4 Byte Integer)
- Double (8 Byte Fliesskomma)
- Null-terminierte Strings
- Pointer(32/64 Bit) auf Arrays der o.g. Typen

Eine eventuell notwendige Umsetzung in framework- oder programmiersprachenspezifische Datentypen wird von den jeweiligen Schnittstellenmodulen übernommen.
Wie in Abbildung 23 dargestellt, greifen die beiden Module der endnutzerorientierten Layer, MOPS Studio und das Excel Add-In nicht direkt auf den Model Management Layer zu, sondern nur indirekt über die COM-Schnittstellen oder alternativ über die .NET-Schnittstellen. Zwar wäre auch ein direkter Zugriff auf den Model Management Layer möglich, die Benut-

zung der COM- oder .NET-Module ist aber einfacher, da diese bereits für den jeweiligen Kontext angepasst sind (z.B. Unterstützung für Excel Datentypen wie `RANGE`).

6.3 Prozedurale Schnittstellen

Dieser Abschnitt beschreibt die Schnittstellenmodule des Procedural Interface Layers. Alle Module nutzen für die Modellverwaltung den Model-Manager und leiten einen großen Teil der Aufrufe ihrer eigenen Funktionen, teilweise modifiziert, an das PMML-Interface weiter.

6.3.1 Kommandozeilenschnittstelle

Die Kommandozeilenschnittstelle ist eine Programmier- und eine Benutzerschnittstelle zugleich, wobei die Programmierung des Executables mit Batch-Scripts oder mit Interprozesskommunikation (z.B. via Pipes) erfolgen kann. Für die Einbindung in EUS ist die praktische Bedeutung der Kommandozeilenschnittstelle gering, da sie eher zu Test- und Demonstrationszwecken genutzt wird. Ihr Vorteil ist die sofortige Lauffähigkeit ohne vorherige Installation und die Plattform übergreifende Invarianz, denn praktisch alle Systemplattformen verfügen über einen Kommandozeilenmodus. Zwar beschränkt sich diese Arbeit auf Windows als Betriebssystem, falls aber auch andere Plattformen einbezogen werden sollten, so ginge der einfachste Weg über die Portierung der Kommandozeilenschnittstelle. Aus diesem Grund soll hier am Beispiel der Kommandozeilenschnittstelle von MOPS (MOPS.EXE) eine Erweiterung deren Spezifikation des diskutiert werden. Das bisherige Kommandozeileninterface von MOPS sieht den folgenden Aufruf vor, um das im *Profile* angegebene Modell mit den dort gesetzten Parametern zu optimieren:

```
> mops [name_of_profile]
```

Eine Erweiterung könnte nun darin bestehen, ein textbasiertes Kommandozeileninterface zu entwickeln, bei dem der Benutzer z.B. durch das Starten der MOPS.EXE in einen speziellen Prompt-Modus gelangt, in dem er das Modell interaktiv eingeben kann. Nach diesem Prinzip funktionieren beispielsweise die AMPL-Kommandozeilenschnittstelle oder der *CPLEX Interactive Optimizer*. Da auf diese Weise nur sehr kleine Modelle bearbeitet werden können, der Implementationsaufwand aber recht hoch ist, dürfte der Nutzen einer solchen Lösung allerdings sehr gering sein.

Sinnvoller erscheint es, neben der bisherigen Methode des Ladens von Profiles, das Laden eines Script Files vorzusehen. Solche Scripts können als *MOPS Scripts* bezeichnet werden und würden als sehr einfache Scriptingsprache dazu dienen, die grundlegenden Operationen

des Optimierers zu steuern. Das Entscheidende dabei ist, dass MOPS Script Plattform unabhängig einsetzbar wäre. Der Aufruf könnte erfolgen mit:

```
> mops /s name_of_script_file
```

Die Option `/s` zeigt an, dass die folgende Datei kein Profile, sondern ein Script ist. Das Script File ist eine Textdatei, die von einem Parser im Kommandoschnittstellenmodul durchgegangen wird. Sie enthält Schlüsselwörter mit Argumenten, die die Ausführung von einigen einfachen Operationen bewirken. Tabelle 15 zeigt einen Basissatz an Befehlen, der ausreicht, um Modelle in verschiedenen Formaten zu lesen, Parameter zu setzen, einen Optimierungslauf zu starten, einfache Ein- und Ausgaben zu machen und Daten über Umgebungsvariablen auszutauschen.

MOPS Script-Befehle
`ReadModel (FileName, FormatSpecifier)`
`WriteModel (FileName, FormatSpecifier)`
`WriteSolFile (FileName)`
`ReadParamFile (FileName)`
`Optimize`
`SetDblParameter (ID, Wert)`
`SetIntParameter (ID, Wert)`
`SetStrParameter (ID, Wert)`
`GetParameter (ID, $Variable)`
`Set $Variable = Wert`
`Print (Wert)`
`Read ($Variable)`

Tabelle 15: MOPS Script-Befehle

Komplexere Scripts sind damit mangels Programmablaufsteuerungsbefehlen (If...Then, Loops etc.) freilich nicht möglich. Solche Features ließen sich zwar grundsätzlich hinzufügen, allerdings würde das einen erheblichen Implementationsaufwand mit sich ziehen. Ein derart aufwändiger Ausbau zu einer vollständigen Scriptingsprache erscheint allerdings angesichts des begrenzten Einsatzgebiets als unangemessen.

6.3.2 Statische Libraries

Die grundsätzliche Konzeption der statischen Interfaces auf dem Procedural Interface Layer stellt keinerlei Schwierigkeiten dar: Es müssen lediglich die Funktionen des Procedural Model Management Layer Interfaces (PMMLI, Anhang 6) gekapselt und in Form einer statisch einbindbaren Bibliothek für die Applikationsentwicklung zur Verfügung gestellt werden. Eine statische Einbindung kommt vor allem für die Programmiersprachen C und Fortran in Frage. Für C müssen die exportierten Funktionen in einem Header deklariert werden. Bei Fortran ist dies in Form einer Interfacedeklaration auch möglich, aber nicht zwingend. Einziger Unterscheid zwischen den LIB-Versionen für C und Fortran ist die unterschiedliche String-

Repräsentation in beiden Sprachen: Während C-Strings nullterminiert sind, besitzen Fortran-Strings eine explizite Längenangabe, die beim Aufruf von Routinen mit Stringargumenten übergeben wird (vgl. [Adams02]). Dieser Unterschied muss beim Design der C- und Fortran-Wrapper für die jeweiligen LIB-Versionen berücksichtigt werden, so dass Aufrufe der LIB mit dem jeweils nativen String-Format der Programmiersprache erfolgen können.
Schwieriger als diese reißbrettartige Neukonzeption ist es, die bestehenden prozeduralen statischen Schnittstellen des als Referenzsystem gewählten Solvers MOPS graduell so umzuformen, dass sie dem hier skizzierten Schnittstellenkonzept nahe kommen. Im Sinne eines Zwischenschritts musste dabei die historisch gewachsene statische Schnittstelle von MOPS, das so genannte IMR-Interface, innerhalb des bestehenden Systems modifiziert werden. Ziel dieses Zwischenschritts war zunächst die Vereinheitlichung der prozeduralen Schnittstellen, indem die IMR-Schnittstelle der bestehenden DLL-Schnittstelle von MOPS angenähert wurde. So wurde begleitend zu dieser Arbeit eine Reihe von Verbesserungen an der bestehenden statischen Bibliothek (MOPS.LIB) vorgenommen, die in den folgenden Abschnitten erläutert werden sollen.

6.3.2.1 IMR-Schnittstelle in MOPS 6.x

Die Routinen der MOPS.LIB entsprachen nur grob den äquivalenten Funktionen der DLL (siehe Dokumentation von MOPS 6.x in [Suhl04]). Beispielsweise gab es eine LIB-Funktion `xnwcol(j,mode,name,len,type,cj,lb,ub,b)`, die analog zur DLL-Funktion `PutCol(j,mode,name,type,cj,lb,ub)` eine Spalte hinzufügte. Es gab aber auch Funktionen, die kein Äquivalent in der DLL hatten, wie z.B. `xgnimr` zur Erzeugung einer internen Modelldarstellung aus Triplets. Dem Benutzer bzw. Programmierer boten sich somit zwei zwar verwandte, aber keineswegs gleichartige Schnittstellen.
Die Schnittstelle der MOPS.LIB wird in der Dokumentation auch als *IMR-Schnittstelle* bezeichnet, da der Benutzer Zugriff auf die *Interne Modell-Repräsentation (IMR)* hat, was bei der MOPS.DLL nicht der Fall ist. Abbildung 24 schematisiert einen Teil der MOPS IMR. Die Elemente der Modellmatrix werden lückenlos spaltenweise in den Arrays `xa`, `xia` und `xjcp` gespeichert, wobei `xa` die Koeffizienten und `xia` die dazugehörigen Zeilenindizes enthält. Die Elemente des Vektors `xjcp` enthalten Zeiger auf den Beginn und das Ende aller Spalten in `xa/xia`. Zielfunktionskoeffizienten sowie Ober- und Untergrenzen speichern die Arrays `xcost`, `xlb` und `xub`. Die Zeilen- und Spaltennamen sind im Array `xrcnam` gespeichert, mit Zeigern in `xrcptr` auf das jeweils erste Zeichen des Namens und der Stringlänge in `xrclen`.

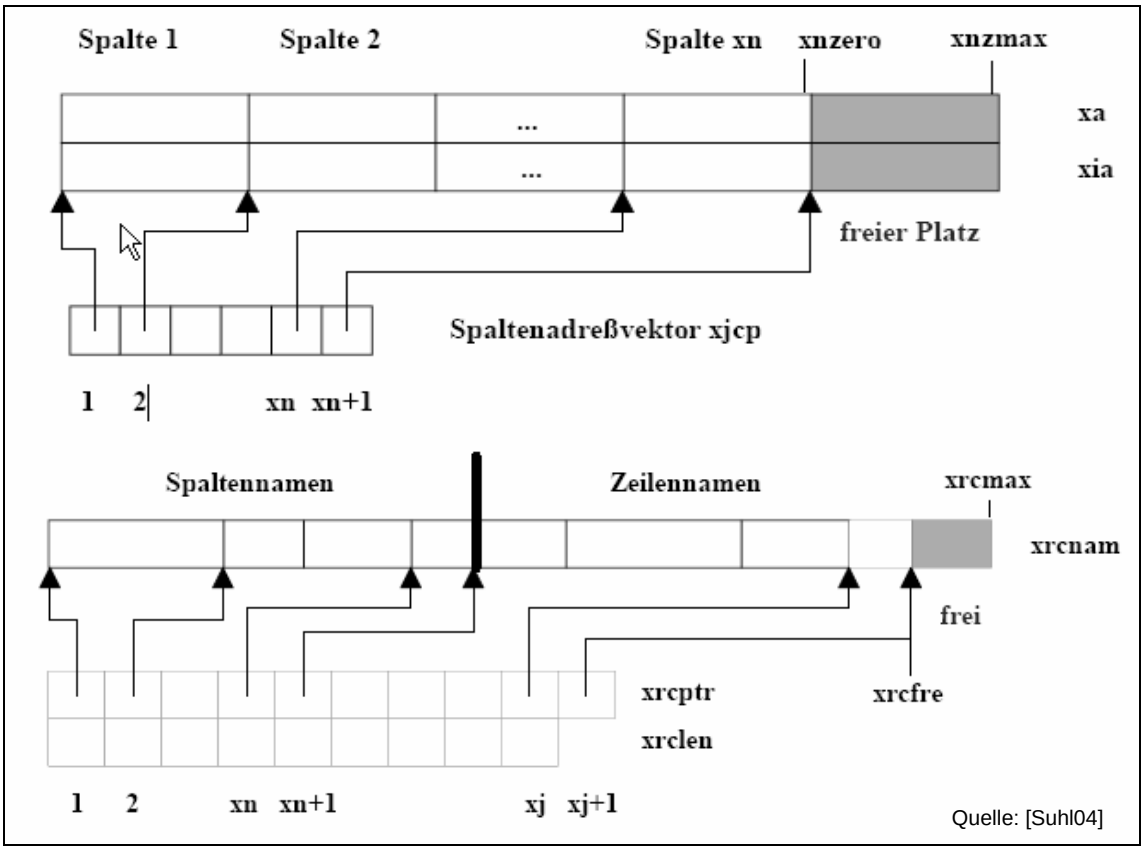

Abbildung 24: MOPS IMR

Diese und weitere MOPS-interne Arrays sind Teil eines großen, zuvor angelegten Speicherbereichs, aus dem die für das Modell benötigten Arrays „herausgeschnitten“ werden. Ein Pointer `b` auf den Anfang dieses globalen Speicherbereiches musste vor Umgestaltung der Bibliothek beim Aufruf der meisten IMR-Funktionen als Argument mitgegeben werden, beispielsweise im IMR-Funktionsaufruf `xptobj(j,cj,b)` zum Setzen eines Zielfunktionskoeffizienten `cj` am Index `j`. Bei anderen, tiefer gelegenen IMR-Funktionen mussten diverse Array-Pointer (z.B. auf `xa`, `xia` usw.) mitgegeben werden, was die IMR-Funktionsaufrufe entsprechend aufblähte.

Durch Einbindung der Deklarationen des Common Blocks `xcr` hat der Fortran-Programmierer vollen Zugriff auf insgesamt weit über 500 globale MOPS-interne Variablen und Arrays. So kann ein Modell auch ohne Benutzung der IMR-Modellgeneratorfunktionen direkt auf den internen Speicherstrukturen erstellt werden. Ebenso kann man auf alle relevanten MOPS-Parameter direkt zugreifen.

Auf diese Art lässt sich die LIB jedoch nur aus Fortran benutzen, da die Einbindung des Common Block Codes unabdingbar ist. Die Nutzung aus C/C++ ist über einen komplizierten Interlanguage-Mechanismus zwar möglich, aber ungleich schwieriger: Zunächst ist ein Bibliotheksformat kompatibler C-Compiler erforderlich, dann müssen die C-Calling-Conventions

auf das Format der Fortran-LIB angepasst werden, unterschiedliche Datentypen (vor allem Strings) müssen umgewandelt werden, und schließlich benötigt man C-Pointer für alle MOPS-Arrays und -Variablen die im Modellgenerator benutzt werden. Weitere Details gibt Kapitel 7.1.

6.3.2.2 IMR-Schnittstellen ab MOPS 7.x

Angesichtes der beschriebenen Schwierigkeiten sollte zum einen die Benutzung der IMR-Schnittstelle vereinfacht, und zum anderen die historisch gewachsene Architektur und Programmstruktur der Interfaceroutinen bereinigt werden. Die hierzu durchgeführten Änderungen setzen das eingangs beschriebene Gesamtkonzept (Layer, Model Manager etc.) noch nicht um, können aber als erster Zwischenschritt dahin gesehen werden.
Zunächst wurden die Aufrufe der IMR-Funktionen vereinfacht, indem die Notwendigkeit der Übergabe interner Pointer eliminiert wurde. Weiterhin wurde die Architektur der IMR-Schnittstelle durch ein verbessertes Pointer-Management gestrafft, so dass eine komplette Aufrufzwischenebene wegfiel. Dies diente dazu, die Benutzung der IMR-Schnittstelle der DLL anzunähern und Implementationsdetails vor dem Benutzer zu verbergen, wobei der Zugriff auf alle MOPS-internen Variablen und Datenstrukturen des Common Blocks weiterhin möglich ist.
Eine andere Verbesserung sowohl der DLL-, als auch der IMR-Schnittstelle war die Erweiterung beider Schnittstellen um zusätzliche Funktionen, die einen feineren Zugriff auf Bestandteile des Modells ermöglichen und als Vorarbeit für die Anbindung an das Modellierungssystem AIMMS galten. Tabelle 16 führt die neuen IMR-Funktionen mit der jeweils analogen DLL-Funktion auf. Bis auf den Namen sind die Funktionen bezüglich ihrer Argumente und Rückgabewerte gleich. Lediglich zu DLL-Funktionen, die den Wert einer einzelnen internen MOPS-Variable liefern, gibt es keine analoge IMR-Funktion, da der Benutzer der statischen Library über den Common Block direkten Zugriff auf diese Variablen hat.

DLL-Funktionen	IMR-Funktionen
ChangeColLB	Xchclb
ChangeColType	Xchctype
ChangeColUB	Xchcub
ChangeCost	Xchcost
ChangeNonzeros	Xchnzs
ChangeLhs	Xchlhs
ChangeRhs	Xchrhs
GenFileNames	Xgenfn
GetColBase	Xgtcbs
GetColIPSolution	Xgtxx
GetColLB	Xgtclb
GetColLPSolution	Xgtxx
GetColUB	Xgtcub

GetColType	Xgtctype
GetCost	Xgtcost
GetIPObjValue	-
GetLPObjValue	-
GetNumberOfColumns	-
GetNumberOfNZ	-
GetNumberOfRows	-
GetRedCost	Xgtrco
GetRowBase	Xgtrowbs
GetRowDualValues	Xgtrco
GetRowLhs	xgtrowlhs
GetRowIPSolution	Xgtxx
GetRowLPSolution	Xgtxx
GetRowRhs	xgtrowrhs
GetRowType	xgtrowtype
GetSolutionStatus	-

Tabelle 16: MOPS DLL- und IMR-Funktionen

Neben der Vereinfachung der IMR-Funktionsaufrufe und den zusätzlichen Funktionen wurde als Drittes das C-Interlanguage-Interface verbessert. Ein „Übereinanderlegen“ von Fortran Common Block und einer passgleichen C-Struktur brachte vollen Zugriff von C auf alle globalen MOPS-Variablen, ohne dass – wie zuvor – für MOPS-Variablen und -Arrays jeweils einzelne C-Pointer anzulegen wären. (Implementierungsdetails in Kapitel 7.1).

6.3.2.3 Konzept zur Weiterentwicklung

Wie bereits oben erwähnt, müsste MOPS zur Einbindung in das Middleware-Konzept – ebenso wie andere Solver auch – nur über das Kernel-Interface angesprochen werden und die LIBs für Fortran und C enthielten Kapselungen der in Anhang 6 aufgelisteten Funktionen. Dieser Funktionssatz ist allerdings neu entworfen, und es muss berücksichtigt werden, dass es unter Umständen Altanwendungen gibt, die den in Tabelle 16 aufgeführten nativen MOPS-Funktionssatz benötigen.

Hierzu ließe sich ein Interface-Modul erstellen, das nach außen die alte IMR-Schnittstelle emuliert, aber intern dem neuen Konzept folgt, also ein Modell mithilfe des Model-Managers erzeugt und MOPS über das Kernel-Interface anspricht. Schwierig wäre dabei allerdings die Umsetzung von Zugriffen auf die MOPS-internen Datenstrukturen. Die entsprechenden Arrays und Variablen müssten ebenfalls emuliert werden, d.h. sie müssten dem Programmierer/Benutzer zur Verfügung gestellt werden und vor Optimierung intern zu einem Modell transformiert werden, das dem Model-Manager übergeben werden könnte. Obwohl programmiertechnisch möglich, wäre ein solcher Aufwand faktisch allerdings kaum gerechtfertigt, da die allermeisten Anwendungen die DLL-Schnittstelle nutzen und keine größere Kundenanwendung MOPS statisch einbindet.

Auf ein Interlanguage Interface der bisherigen Form sollte ganz verzichtet werden, da wegen des zwischengeschalteten Model Management Layers ein direkter Zugriff auf den Common Block und die IMR nicht möglich ist und zudem das Arbeiten mit MOPS-internen Datenstrukturen dem Programmierer eines Modellgenerators keinerlei ersichtlichen Vorteil bringt. Der Performancegewinn dürfte kaum messbar sein, und der Verzicht auf Datenverifikation durch die LIB-Funktionen birgt ein größeres Fehlerrisiko. Ein C-spezifisches statisches Bibliotheksmodul auf dem Procedural Interface Layer brächte nicht nur eine einfachere Benutzung, es wäre durch den Verzicht auf Pointervariablen auf Common Block-Bereiche auch erheblich fehlerresistenter.
Schließlich sei noch die Möglichkeit der Erstellung von statischen Kompatibilitätsbibliotheken genannt, die den Einsatz eines weniger verbreiteten Optimierers, wie z.B. MOPS, in Modellgeneratorcode ermöglichen, der ursprünglich für andere Solver geschrieben wurde. So existiert eine Vielzahl von Generatoren, die kommerzielle statische Solverbibliotheken, wie z.B. CPLEX (cplex101.lib) oder XPRESS (xprs.lib) nutzen. Eine Kompatibilitätsbibliothek emuliert ein Subset des Funktionsumfangs einer solchen Solverbibliothek, indem die implementierten Funktionen exakt die gleichen Funktionsnamen, Argumente und Calling Sequences benutzen. Der Anwender braucht dann lediglich beim Linking die ursprüngliche Library durch die entsprechende Kompatibilitätsbibliothek auszutauschen, während Header und Sourcecode unverändert bleiben. Auf diese Art lassen sich die grundlegenden Routinen diverser anderer Solverbibliotheken emulieren – jedoch keineswegs alle Funktionen. Insbesondere Callbacks, Cuts und andere Lösungsprozessteuermechanismen sind in der Regel zu verschieden und zu solverspezifisch, als dass eine Übersetzung möglich wäre. Auch können bei Weitem nicht alle Parameter übertragen werden. Da eine statische Kompatibilitätsbibliothek den gleichen Funktionsumfang wie ihr DLL-Pendant hätte, soll als Beispiel auf die in 6.3.4 beschriebene dynamische Kompatibilitätsbibliothek für XPRESS verwiesen werden.

6.3.3 Dynamische Library

Für viele Solver ist deren bestehende prozedurale DLL-Schnittstelle das wichtigste Interface, daher dürfte auch im hier entworfenen Middleware-Konzept das dynamische Schnittstellenmodul des prozeduralen Layers von großer praktischer Bedeutung sein. Der entscheidende Vorteil einer DLL ist deren flexible Austauschbarkeit ohne Neukompilation der Anwendung.
Die Funktionen der hier konzipierten DLL-Schnittstelle stimmen weitgehend mit denen des Procedural Model Management Layer Interfaces (PMMLI, s. Anhang 6) überein, was eine äußerst schlanke Implementation der DLL-Kapselschicht erlaubt, da fast alle Funktionen ein-

fach an die Schicht des Model-Managers weitergereicht werden können. Die rund 100 Funktionen des DLL-Interfaces stellen einen Durchschnitt der prozeduralen Funktionen verbreiteter State-of-the-art-Solver (Anhang 1) dar, wobei insbesondere die Features von MOPS als Referenzsystem für die Implementation beachtet wurden. Falls erforderlich, ist eine Erweiterung der Schnittstelle um zusätzliche Funktionen jederzeit möglich. So könnten – um nur ein Beispiel zu nennen – Funktionen hinzugefügt werden, die ein Einfügen von Zeilen und Spalten inklusive ihrer Nonzeros ermöglichen, ähnlich der CPLEX-Routinen `CPXaddcols` und `CPXaddrows`. Die Umsetzung einer solchen Funktion in PMMLI-Aufrufe würde dann innerhalb des DLL-Schnittstellenmoduls auf dem Procedural Interface Layer erfolgen.

Was die Benennung der exportierten DLL-Funktionen anbelangt, so sollten diese (wie auch bei der LIB) mit einem spezifischen Präfix versehen werden, um so in Sprachen, die keine Namespaces unterstützen, einen Pseudo-Namensraum für die Optimierungsbibliothek herzustellen und Namenskollisionen zu vermeiden. So könnte beispielsweise eine Funktion `AddCol` als `MOAddCol` (MO für „MOdeler“) exportiert werden.

Parameter werden als symbolische Konstanten definiert, wobei einige Parameter universeller Natur sind, d.h. unabhängig vom eingesetzten Solver eine Bedeutung haben (z.B. die Anzahl der Restriktionen), während andere Parameter solverspezifisch sind (z.B. Wahl einer IP-Heuristik in MOPS). Dementsprechend müssen allgemeine und solverspezifische Parameter definiert werden. Für C/C++ geschieht dies über entsprechende Header, etwa:

```
#define MO_NumberOfRows    123    // allgemeiner Modeler-Parameter
#define MOPS_XRDUCE       4711    // MOPS-spezifischer Parameter
#define CPX_PARAM_ADVIND  1001    // CPLEX-spezifischer Parameter
```

Diese Handhabung von Parametern gilt nicht nur für die DLL-Schnittstelle, sondern auch für die anderen prozeduralen und objektorientierten Interfaces.

Falls ein Solver wie MOPS für das Funktionieren in bestehenden Anwendungen die Funktionen des bisherigen DLL-Interfaces benötigt, wäre dies dank der modularen Ausrichtung des Middleware-Konzepts mit vertretbarem Aufwand realisierbar. In diesem Fall müsste ein zusätzliches Modul auf dem Procedural Interface Layer hinzugefügt werden, das neben den allgemeinen Standard-Funktionen auch die speziellen MOPS-Funktionen der alten MOPS.DLL exportiert. Dabei würde es sich allerdings um eine Emulation des alten Interfaces handeln, denn im Inneren würden Model-Manager und Solver-Manager benutzt.

6.3.4 Kompatibilitätsbibliotheken

Wie bereits mehrfach erwähnt, dienen Kompatibilitätsbibliotheken dazu, die Schnittstelle einer Optimierungsbibliothek zumindest teilweise zu emulieren und so den Einsatz eines fremden Optimierers in Anwendungen zu ermöglichen, die ursprünglich für einen anderen Optimierer entwickelt wurden. In aller Regel ist aber eine vollständige Emulation nicht möglich, da entweder bestimmte Features in dem fremden Optimierungssystem nicht vorhanden oder so unterschiedlich implementiert sind, dass eine Übersetzung inhaltlich nicht machbar ist. Beim Entwurf einer Kompatibilitätsbibliothek kann daher lediglich versucht werden, ein möglichst großes Subset der Funktionen des emulierten Systems abzudecken.

Als Beispiel soll hier das Konzept einer Kompatibilitätsbibliothek für das Callable Library Interface (DLL oder LIB) des XPRESS Solvers umrissen werden, bei der MOPS als Fremdsystem eingesetzt wird und die Optimierungsmiddleware die übersetzende Rolle spielt. Die XPRESS Callable Library eignet sich hierfür besonders gut, da sie ein ähnliches Layout wie die prozeduralen Schnittstellen der Middleware hat. Ihr Funktionsumfang ist nicht so ausufernd wie bei CPLEX, und es existieren nicht so viele solverspezifische Sonderfunktionen (siehe Anhang 1). Die folgende Tabelle 17 unterscheidet alle Funktionen der prozeduralen XPRESS-Solverbibliothek, je nachdem, ob sie von einer Kompatibilitätsbibliothek mit MOPS als Solver implementiert werden könnten oder nicht.

Modellhandling

Übertragbare Funktionen	Nicht übertragbare Funktionen
XPRSinit	XPRScopycallbacks
XPRSfree	
XPRScreateprob	
XPRSdestroyprob	
XPRScopyprob	
XPRScopycontrols	

File I/O

Übertragbare Funktionen	Nicht übertragbare Funktionen
XPRSreadprob	XPRSreadbasis
XPRSrestore	XPRSwritebasis
XPRSwriteprob	XPRSreaddirs
XPRSsave	XPRSiis
XPRSwritesol	XPRSreaddirs
XPRSwriteprtsol	XPRSrange
XPRSwriteomni	XPRSwriterange
XPRSsetlogfile	XPRSwriteprtrange
XPRSalter	

Modellaufbau und -modifikation

Übertragbare Funktionen	Nicht übertragbare Funktionen
XPRSaddcols	
XPRSaddrows	
XPRSchgcoef	
XPRSchgmcoef	
XPRSloadlp	
XPRSloadglobal	

XPRSdelrows	
XPRSdelcols	
XPRSaddnames	
XPRSchgrowtype	
XPRSchgrhs	
XPRSchgrhsrange	
XPRSaddnames	
XPRSchgcoltype	
XPRSchgbounds	
XPRSchgobj	
XPRSaddsetnames	
XPRSchgobj	
XPRSsetprobname	

Modellabfrage

Übertragbare Funktionen	Nicht übertragbare Funktionen
XPRSgetnames	XPRSgetdirs
XPRSgetcoltype	
XPRSgetlb	
XPRSgetub	
XPRSgetobj	
XPRSgetnames	
XPRSgetrowtype	
XPRSgetrhs	
XPRSgetrhsrange	
XPRSgetrows	
XPRSgetcols	
XPRSgetprobname	
XPRSgetindex	
XPRSgetglobal	

Lösungsabfrage und -analyse

Übertragbare Funktionen	Nicht übertragbare Funktionen
XPRSgetsol	XPRSiis
XPRSgetbasis	XPRSgetiis
XPRSgetdblattrib	XPRSrange
XPRSgetintattrib	XPRSrhssa
XPRSgetstrattrib	XPRSgetcolrange
	XPRSgetrowrange

Lösungsprozesssteuerung

Übertragbare Funktionen	Nicht übertragbare Funktionen
XPRSmaxim	XPRSloadbasis
XPRSminim	XPRSaddcuts
XPRSglobal	XPRSloadmodelcuts
	XPRSloadcuts
	XPRSdelcpcuts
	XPRSdelcuts
	XPRSpivot
	XPRSbtran
	XPRSftran
	XPRSgetdirs
	XPRSloaddirs
	XPRSloadpresolvedirs
	XPRSloadpresolvebasis
	XPRSscale
	XPRSpresolvecut
	XPRSloadsecurevecs
	XPRSdelnode
	XPRSfixglobal
	XPRSgetcpcuts
	XPRSgetcpcutlist
	XPRSgetcutlist
	XPRSstorecuts
	XPRSinitglobal

Parameterhandling

Übertragbare Funktionen	Nicht übertragbare Funktionen
XPRSgetdblcontrol	
XPRSgetintcontrol	
XPRSgetstrcontrol	
XPRSsetdblcontrol	
XPRSsetintcontrol	
XPRSsetstrcontrol	
XPRSsetdefaults	

Callbacks

Übertragbare Funktionen	Nicht übertragbare Funktionen
XPRSsetcblplog (teilw.)	XPRSsetcbchgbranch
XPRSsetcbmessage (teilw.)	XPRSsetcboptnode
XPRSsetcbintsol (teilw.)	XPRSsetcbchgnode
	XPRSsetcbbarlog
	XPRSsetcbcutlog
	XPRSsetcbestimate
	XPRSsetcbgloballog
	XPRSsetcbinfnode
	XPRSsetcbnodecutoff
	XPRSsetcbprenode
	XPRSsetcbsepnode
	XPRSsetbranchcuts
	XPRSsetcbcutmgr
	XPRSsetcbinitcutmgr
	XPRSsetcbfreecutmgr
	XPRSstorebounds
	XPRSsetbranchbounds

Fehlermeldungen

Übertragbare Funktionen	Nicht übertragbare Funktionen
XPRSgetlicerrmsg	
XPRSgetlasterror	

Sonstige Funktionen

Übertragbare Funktionen	Nicht übertragbare Funktionen
XPRSgetdaysleft	
XPRSgetbanner	
XPRSgetversion	

Tabelle 17: Funktionen XPRESS-Kompatibilitätsinterface

Die nicht übertragbaren Funktionen stammen vor allem aus dem Bereich „Callbacks“ und „Lösungsprozessteuerung“. MOPS unterstützt erst seit Version 9.x Callbacks, wobei weit weniger Ereignisse einen Callback auslösen können als bei XPRESS. Auch in anderen Details unterscheiden sich die beiden Callback-Konzepte (z.B. Parameter/Argumente der Callbackfunktionen), weshalb eine Übersetzung nur teilweise möglich ist. Ebenso können in MOPS keine Branching Directions und Prioritäten für Integervariablen angegeben werden oder benutzerdefinierte Cuts während des Lösungsprozesses hinzugefügt werden. Weiterhin sind Funktionen zur Range- und IIS-Analyse nicht übertragbar, da MOPS derartige Funktionen gegenwärtig nicht enthält. Für den Warmstart von einer gespeicherten Basis gibt es in XPRESS die Funktionen `XPRSreadbasis` und `XPRSloadbasis`; in frühren MOPS-Versionen war ein solcher Warmstart ebenfalls möglich, ist aber aufgrund der immer schnel-

leren LP-Lösungszeiten heute kaum noch praxisrelevant und wird in der aktuellen MOPS v9.x auch nicht mehr unterstützt, weshalb die genannten XPRESS-Funktionen auch kein Pendant bei MOPS haben und somit nicht übertragbar sind.
XPRESS setzt und liest fast alle einzelnen Werte, seien es Parameter zur algorithmischen Steuerung oder seien es Modell-Attribute, über universelle Funktionen für Double-, Integer- und String-Werte, was in der überarbeiteten DLL-/LIB-Schnittstelle von MOPS v9.x genauso ist. Daher können zwar nicht alle, aber viele XPRESS-Parameter recht einfach vom Solver Manager an MOPS weitergeleitet werden. Andere XPRESS Parameter wie z.B. `ROWS`, `COLS`, `ELEMS` oder `LPOBJVAL` werden direkt vom Modelmanager verwaltet, da sie solverunabhängig das Modell betreffen.
Abschließend sei noch bemerkt dass die Erstentwicklung eines Kompatibilitätsschnittstellenmoduls am besten mit Bezug auf einen konkreten Modellgenerator eines potenziellen Nutzers erfolgt. So können zunächst nur diejenigen Funktionen implementiert werden, die tatsächlich in diesem Generator vorkommen. Nach entsprechenden Tests und Vergleichsläufen der beiden Optimierer liegt eine kleine, aber stabile Funktionsbasis vor, die später um weitere Funktionen ergänzt werden kann.

6.4 Objektorientierte Schnittstellen

Auf dem *Objektorientierten Interface Layer* befinden sich Schnittstellenmodule zur Benutzung aus verschiedenen objektorientierten Sprachen und Frameworks. Spezifiziert werden im Folgenden Interfaces für C++, COM und .NET, durch das modulare Konzept sind darüber hinaus aber auch Schnittstellen für weitere OO-Sprachen realisierbar. Der entscheidende Unterschied zu den prozeduralen Interfaces sind zusätzliche Modellierungsobjekte, die das objektorientierte Schnittstellenkonzept zur Unterstützung der Modellgeneratorprogrammierung vorsieht.

6.4.1 Gemeinsame Modellierungsklassen

Die Gemeinsamen Modellierungsklassen sind eine zum Model Management Layer gehörende Klassenbibliothek, die von den Modulen des OO-Interface Layers zur gemeinsamen Wiederverwendung genutzt werden. Die Interfacemodule des OO-Layers bieten dem Benutzer unterschiedliche, auf dessen Programmier- und Frameworkkontext hin angepasste Schnittstellen, die ihrerseits intern auf einem gemeinsamen „Klassen-Baukasten“ basieren. Der Endnutzer arbeitet dabei nicht mit diesem internen Klassen-Baukasten, sondern mit den Komponenten (COM, .NET) oder Klassen (C++) des darüber liegenden OO-Layers. Die Benutzung der gemeinsamen Modellierungsklassen ist für die Module des OO-Layers optional: Zwar sehen die hier konzipierten C++-, COM- und .NET-Endnutzermodule die interne Verwendung der gemeinsamen Klassen vor, andere Module sind dazu aber keineswegs verpflichtet und können ihre Funktionalität mit eigenen Klassen und Routinen implementieren. Bindend ist lediglich die Solver-Benutzung über den Model Management Layer.

Bei den Gemeinsamen Modellierungsklassen handelt es sich um C++-Klassen, die in einer Implementation als unmittelbare Basis für die C++-Endnutzerklassenbibliothek dienen können. Auch eine in C++ implementierte COM-Komponentenbibliothek kann diese gemeinsamen C++-Klassen verwenden, wobei das COM-Modul wegen der Verschiedenheit der C++- und COM-Schnittstellenstandards allerdings größere Umsetzungen durchführen muss. Desgleichen kann eine .NET-Komponentenbibliothek auf den internen C++-Klassen beruhen. Realisierbar ist dies durch die Verwendung von *Managed C++* (bzw. *Visual C++ 2005*). In dieser Sprache lassen sich managed .NET-Code und der native Code der gemeinsamen C++-Klassen vermischen, so dass ein dergestalt konstruiertes .NET-Schnittstellenmodul nach außen ein CLR-konformes Interfacesystem zur Benutzung aus .NET-Sprachen wie VB.NET und C# bietet.

Das Design der gemeinsamen Modellierungsklassen resultiert aus folgenden Zielsetzungen:

- Vermischbarkeit von objekt- und matrixorientierter Modellierung
- Unterstützung der Modellierung mit überladenen Operatoren (C++, C#)
- Unterstützung einfacher und mehrfacher alphanumerischer Indizes
- Verfügbarkeit alphanumerisch indizierter Datenarrays
- Indizierte Variablen- und Restriktionssets
- Vermeidung von Schleifen durch „Burst Operationen“ für ganze Objektsets

Abbildung 25 zeigt das grobe Schema der Gemeinsamen Modellierungsklassen, und Anhang 4 enthält eine detaillierte Auflistung ihrer öffentlichen Methoden und Eigenschaften in UML-ähnlicher Notation.

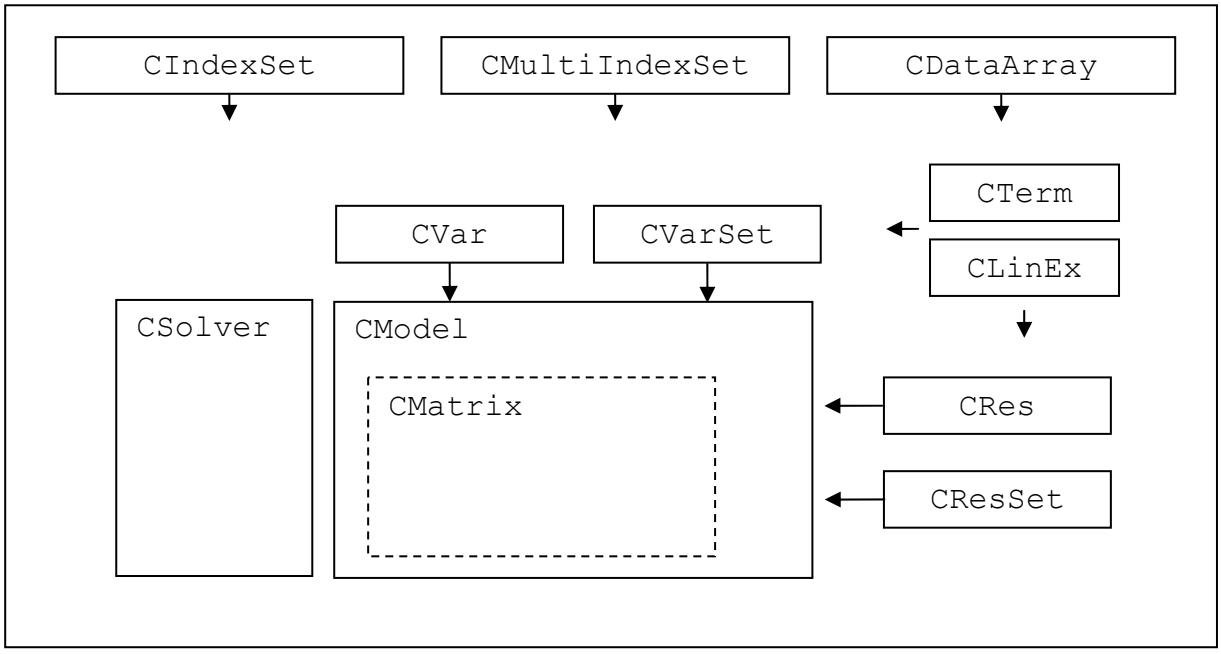

Abbildung 25: Die wichtigsten gemeinsamen Modellierungsklassen

6.4.1.1 CMatrix, CModel und CSolver

Im Mittelpunkt steht die Klasse `CModel`, die das Modell repräsentiert. Die Elternklasse von `CModel` ist `CMatrix`, die prozeduralen Funktionen des PMML-Interfaces in objektorientierter Form widerspiegelt. Wie auch bei den prozeduralen Schnittstellen, übernimmt der Modell-Manager die eigentliche Modellverwaltung und übergibt das Modell vor Optimierung an den Solver-Manager, der durch die Klasse `CSolver` dargestellt wird.

`CModel` und `CMatrix` repräsentieren aber gleichzeitig auch zwei unterschiedliche Sichten auf das Modell: Einerseits die matrixorientierte, aus Zeilen, Spalten und Nonzeros bestehende Sichtweise von `CMatrix` und andererseits die modellierungsobjektorientierte Sichtweise von

`CModel` und den Modellierungsklassen, die durch die Abbildung von Variablen, Restriktionen, Sets, linearen Ausdrücken etc. näher an die mathematische Notation heranrückt und damit ein höheres Abstraktionsniveau darstellt. Die Ableitungsrelation beider Klassen bringt die Parallelität und gleichzeitige Verwendbarkeit beider Perspektiven zum Ausdruck, denn das besondere an der hier vorgeschlagenen Konzeption ist, dass der Entwickler eines Modellgenerators nicht nur alternativ die eine oder andere Sichtweise annehmen kann, sondern innerhalb des Modellgenerators auch beide Perspektiven beliebig vermischen kann. Das unterscheidet das vorlegende Konzept von anderen Modellierungsbibliotheken. So findet man z.B. in XPRESS ebenfalls beide Perspektiven, nämlich die eher matrixorientierte Perspektive der XPRESS Solver Library sowie die optimierungsobjektbezogene Perspektive der BCL. Dabei sind beide Perspektiven aber an unterschiedliche Modellierungsbibliotheken gekoppelt, und das Modell wird durch zwei unterschiedliche Instanzen dargestellt, die nur teilweise miteinander synchronisiert werden. Im vorliegenden Konzept existiert hingegen nur eine einzige Modellinstanz, nämlich die von `CModel`, weshalb keine Synchronisation zweier Modellrepräsentationen erforderlich ist. Die Verknüpfungen der Modellierungsobjekte mit dem Modell bzw. der Matrix erfolgt für den Benutzer weitgehend transparent. Zwar arbeitet der Benutzer nicht direkt mit den in diesem Kapitel beschriebenen internen Modellierungsklassen, jedoch überträgt sich das Design der internen Klassen auf die Benutzerschnittstellen des OO-Layers. Dies gilt insbesondere im Falle der C++-Benutzerbibliothek, die lediglich einen sehr dünnen Wrapper um die internen Klassen bildet.

6.4.1.2 Einfache Indexsets

Einer der wichtigsten und effizienzrelevantesten Bereiche bei Modellgeneratoren und Modellierungssprachen, ist die Indizierung (vgl. etwa [Geoffrion92], [Hürlimann00], [Kuip92]). Indexsets sind geordnete Mengen von numerischen Elementen oder Strings. Die Indizierung mittels fortlaufender, ganzzahliger Indizes ist das normalerweise in allen Programmiersprachen verwendete Prinzip zum Zugriff auf Arrays. In Modellgeneratoren kommt es jedoch häufig vor, dass Indizes alphanumerischer Natur sind (z.B. Wochentage, Namen von Mischungszutaten oder Namen von Maschinen bei Scheduling-Problemen). Zwar lassen sich solche alphanumerischen Indizes auf ganzzahlige numerische Indizes umsetzen, indem Zuordnungstabellen hinterlegt werden, jedoch sind intuitiv verständliche Zuordnungen wie etwa bei Monatsnamen (Januar → 1, Februar → 2, etc.) eher die Ausnahme. Daher ist ein Modellgenerator, der nur mit numerischen Indizes arbeitet, in der Regel schwerer zu erstellen, im Code weniger intuitiv und stärker fehleranfällig, als wenn auch alphanumerische Indizes verwendet

werden können. Die Benutzung alphanumerischer Indizes soll durch die Klasse `CIndexSet` erleichtert werden. `CIndexSet` ist als Templateklasse in Bezug auf den Elementtyp konzipiert, so dass sich durch entsprechende Instanzierung sowohl numerische (Integer, Double), als auch alphanumerische (CString) Indizes darstellen lassen. Neben Methoden zum Hinzufügen und Löschen von Elementen gibt es Methoden zur Erzeugung arithmetischer Reihen und zum Zugriff auf bestimmte Indexpositionen. Es existieren Methoden zum Kopieren von Indexsets, sowie Mengenoperationen zur Erzeugung von Vereinigungs-, Differenz- und Schnittmengen (`Union`, `Diff`, `Intersect`). Weiterhin ist mit den Methoden `MoveFirst`, `MoveNext`, `GetCurrent` und `IsLast` eine einfache Iteratorfunktionalität implementiert.

6.4.1.3 Multi-Indexsets

In vielen realen Modellen kommen mehrfache Indizierungen vor; etwa bei Supply-Chain-Management-Modellen sind zehn und mehr Indizes keine Seltenheit ([Kallrath04b, S. 66]). Bei mehrfacher Indizierung bilden n Indexmengen das Kartesische Produkt

$$\prod_{i=1}^{n} A_i = A_1 \times ... \times A_n = \{(a_1,...a_n) \mid a_i \in A_i \quad i = 1,...,n\}$$

so dass die n-Tupel (a_1, …, a_n) die Ausprägungen des Mehrfachindex bilden. Die Anzahl m der möglichen n-Tupel ist das Produkt der Mächtigkeit der einzelnen Indexmengen $m=|A_1| \cdot \ldots \cdot |A_n|$. Mehrfach-Indexsets werden durch die Klasse `CMultiIndexSet` dargestellt, die eine beliebige Anzahl von Indexmengen des Typs `CIndexSet` zusammenfasst und das Kartesische Produkt enumerierbar macht. `CMultiIndexSet` verfügt über mehrere überladene Konstruktoren, die der Initialisierung mit den einzelnen Indexsets dienen. Weiterhin gibt es Methoden zum Setzen und Auslesen einzelner Indexsets, die teilweise die Möglichkeit variabler Parameterlisten in C++ ausnutzen.

Wie auch bei einfachen Indexsets sind Iteratormethoden vorgesehen, die über alle n-Tupel des zusammengesetzten Indexsets laufen können. Zusätzlich existiert ein Filtermechanismus für die Iteratoren, mit dem Teile der zu durchlaufenden Tupelmenge ausgeblendet werden können. Dies soll an einem Beispiel verdeutlicht werden: Angenommen, es gibt zwei Objekte (`Set1`, `Set2`) von Typ `CIndexSet`, von dem das erste Jahre und das zweite Monate aufzählt (Set1={2005, 2006, 2007}, Set2={Jan, Feb, Mrz, … Dez}). Beide Sets werden in einem „Perioden“ genannten `CMultiIndexSet` zusammengefasst. Möchte man in einer Schleife beispielsweise lediglich das zweite Quartal 2006 durchlaufen, so müsste folgender Iterator-Filter gesetzt werden:

```
Perioden.SetIterFilter(CIndexSet(2006), CIndexSet("Apr", "Mai", "Jun"))
```

Sollen die zweiten Quartale aller Jahre durchlaufen werden, so gäbe man an:

```
Perioden.SetIterFilter (CIndexSet(), CIndexSet("Apr", "Mai", "Jun"))
```

und würde anschließend mit `MoveFirst`, `MoveNext` etc. über die Tupel iterieren.

Die Klasse `CMultiIndexSetIter` dient ebenfalls der Iteration. Sie läuft über einfache Indexsets und über Multi-Indexsets, ohne eine Kopie von diesen zu enthalten. Mit Iteratoren lassen sich auf eine sehr intuitive Art Schleifen programmieren, wie folgende Formulierung einer For-Schleife zeigt:

```
for(myIter.MoveFirst(); !myIter.IsBeyondLast(); myIter.MoveNext()){…}
```

Die Hilfsklasse `CTuple` repräsentiert eines der *m* möglichen Tupel (a_1, …, a_n) eines `CMultiIndexSets` und bietet Zugriff auf die einzelnen Elemente a_i. Da ein MultiIndexSet-Objekt aus Indexsets unterschiedlichen Typs (String, Double, Integer) zusammengesetzt sein kann, muss die Klasse `CTuple` dem Rechnung tragen, indem die Methoden `SetElement` und `GetElement` für diese Typen überladen sind.

6.4.1.4 Datenarrays

Die meisten Programmiersprachen bieten nur Unterstützung für Arrays mit ganzzahlig-numerischen Indizes, die häufig, wie im Falle C++ oder VB.NET, von Null als fester Untergrenze aus laufen müssen. Diesen Mangel soll die Klasse `CDataArray` beheben, die ein mit Multi-Indexsets indizierbares ein- oder mehrdimensionales Array darstellt. `CDataArray` ist als Templateklasse angelegt, so dass der Typ der enthaltenen Daten erst bei Instanzierung festgelegt wird und damit die drei Grundtypen CString, Integer und Double abgedeckt werden. `CDataArray` ist von `CMultiIndexSet` abgeleitet und enthält daher ein Multi-Indexset, nebst dessen öffentlicher Methoden, das zur Indizierung der Array-Klasse benutzt wird. Initialisiert wird `CDataArray` im Konstruktor oder durch die `Init`-Methoden der Elternklasse `CMultiIndexSet`. Weiterhin existieren Methoden zum Zugriff auf Datenfelder und zum gleichzeitigen Setzen von Werten für mehrere, durch einen Indexset-Filter bestimmte Arraypositionen. Mit dem von der Elternklasse geerbten Iterator kann über alle oder nur über Subsets der Datenfelder iteriert werden, wobei mit `CurrentField` auf das jeweils aktuelle Datenfeld zugegriffen wird.

6.4.1.5 Variablen- und Restriktionsklassen

Die Klassen `CVar` und `CRes` repräsentieren eine einzelne Modell-Variable bzw. eine Restriktion. `CVar` macht Variablenattribute wie Ober- und Untergrenze, Kosten etc. zugreifbar und

änderbar. Ein `CVar`-Objekt muss dazu mit einer konkreten Modellinstanz verbunden sein, und Änderungen im `CVar`-Objekt werden unmittelbar im Modell durchgeführt; es gibt keine parallele Speicherung von Werten im Variablenobjekt und im Modell, die eine Synchronisation erforderlich machen würde. `CVar` ist eine abstrahierte und objektorientierte Repräsentation. Die selben Daten sind aber auch matrixorientiert über entsprechende Methoden von `CModel` bzw. deren Elternklasse `CMatrix` zugreifbar. Selbstverständlich werden Löschungen oder Einfügungen im Modell, die den Spaltenindex des `CVar`-Objekts verändern, intern berücksichtigt, so dass ein verbundenes Variablenobjekt immer auf die richtige Spalte zeigt. Wegen der beliebigen Vermischbarkeit von matrix- und objektbasierter Modellierung müssen keineswegs alle Spalten durch ein `CVar`-Objekt dargestellt sein. Es können aber auch mehrere Variablenobjekte auf ein und dieselbe Spalte zeigen.

Analog zu `CVar` ist die Restriktionsklasse `CRes` die objektorientierte Darstellung einer verbundenen Matrixzeile. Zusätzlich kommen aber bei `CRes` noch Methoden zum Hinzufügen und Entfernen von Variablen, Termen und linearen Ausdrücken hinzu, denn die Modellgenerierung mit den hier aufgeführten Modellierungsobjekten ist restriktionsorientiert. Auf die sehr intuitive Restriktionsformulierung, die mit Termen, linearen Ausdrücken und Operatorüberladung möglich ist, wird im folgenden Kapitel 6.4.2 ausführlicher eingegangen. Schließlich besitzt `CRes` noch einen Iterator-Mechanismus zur Ermöglichung der Iteration über alle Variablen der Restriktion.

Für ein- oder mehrfach indizierte Variablen- und Restriktionsmengen sind die Klassen `CVarSet` und `CResSet` vorgesehen. Diese sind von `CMultiIndexSet` abgeleitet und enthalten daher Methoden für die Verwaltung von (Mehrfach-)Indizes. Zur Initialisierung von `CVarSet` und `CResSet` müssen die zur Indizierung verwendeten (Multi-)Indexsets angegeben werden. Die Benennung entspricht dem Schema der Multi-Indexsets. Beispiel: Eine Variable *X*, die über die Indexmengen A und B mit Tupeln (a_1,b_1), (a_1,b_2),...(a_n,b_m) indiziert ist, wird benannt als "*X.a1.b1*", "*X.a1.b2*", ... "*X.an.bm*". Man kann beliebig auf einzelne Variablen bzw. Restriktionen eines Sets zugreifen, jedoch ist die Veränderung der automatisch generierten Variablen- und Restriktionsnamen nicht gestattet. Ebenso wird verhindert, dass innerhalb eines `CVarSet` oder `CResSet` Variablen oder Restriktionen eingefügt oder gelöscht werden, so dass die Integrität des Blocks immer sichergestellt ist. Beide Klassen bieten Methoden zum Zugriff auf einzelne Variablen oder Restriktionen, die als `CVar` oder `CRes`-Objekte gesetzt oder zurückgegeben werden können. Einzelne Attribute wie LB, UB, RHS etc. können global für das gesamte Set geändert werden, aber auch eine selektive Änderung über Indexset-Filter ist möglich. Dabei funktioniert der Filtermechanismus nach folgendem

Prinzip: Wird ein Indexset *S* als Parameter angegeben, so erfolgt die Veränderung nur an den Elementen der Schnittmenge mit dem Basisindexset *B*; ein nicht angegebenes oder leeres Indexset bewirkt die Einbeziehung aller Elemente des Basisindexsets. Hierzu ein Code-Beispiel:

```
CIndexSet <int> basisset(2005,2006,2007,2008);
CIndexSet <int> filterset(2005,2007,2020);
CVarSet x(mymodel, basisset);
x.Name = "X";
x.SetLBs(4711, filterset);
```

Dies bewirkt, dass ein aus *X.2005, X.2006, X.2007* und *X.2008* bestehendes Variablenset mit Defaultwerten erzeugt wird und dass bei den Variablen *X.2005* und *X.2007* die Untergrenze auf den Wert 4711 gesetzt wird. Vorteil ist ein kompakterer, intuitiverer und fehlerresistenterer Code durch die Vermeidung von Schleifen und `If`-Verzweigungen. Des Weiteren gib es in `CVarSet` und `CResSet` noch die von `CMultiIndexSet` geerbten Iteratormethoden, die über die einzelnen Variablen bzw. Restriktionen iterieren und das jeweils aktuelle Element über die Attribute `CurrentVar` bzw. `CurrentRes` zugreifbar machen. Schließlich können in der Klasse `CResSet`, ähnlich wie bei einer einzelnen `CRes`-Restriktion Variablen, Terme und lineare Ausdrücke hinzugefügt oder gelöscht werden.

6.4.1.6 Lineare Ausdrücke

Die beiden Klassen `CTerm` und `CLinEx` repräsentieren einen Term bzw. einen linearen Ausdruck. `CTerm` ist eine sehr einfache Klasse, die lediglich einen Verweis auf ein `CVar`-Objekt und einen dazugehörigen Koeffizienten speichert. Mehrere Terme und Variablen können zu Objekten vom Typ `CLinEx` verknüpft werden und bilden so einen linearen Ausdruck der Form $c_1 \cdot Var_1 + c_2 \cdot Var_2 + \ldots + c_n \cdot Var_n$. Dementsprechend existieren in `CLinEx` Methoden zum Hinzufügen und Löschen von Variabeln, Termen und linearen Ausdrücken. Auch können ganze Variablensets dem Ausdruck hinzugefügt werden. Wird dabei ein `CDataArray` als Argument für die Koeffizienten angegeben, so extrahiert die Methode die entsprechenden Koeffizienten aus dem `CDataArray`, sofern dieses die gleiche Dimension wie das `CVarSet` hat und seine Indexsets eine Übermenge der Indexsets des `CVarSet` bilden. Beispiel:

```
CIndexSet <CString> basisset("Jan","Feb","Mrz");
CVarSet x(mymodel, basisset);
x.Name = "X";
```

```
CDataArray <double> da(basisset);
da.SetElementAt(4711, "Jan");
da.SetElementAt(4712, "Feb");
da.SetElementAt(-4713, "Mrz");
CLinEx lx(x,da);
```

Das Codebeispiel ergibt den linearen Ausdruck `lx`

```
4711·X.Jan + 4712·X.Feb – 4713·X.Mrz
```

welcher zur Formulierung von Zielfunktion oder Restriktionen verwendet werden kann.

6.4.2 C++-Klassenbibliothek

Die C++-Klassenbibliothek dient der objektorientierten Einbindung der Optimierungsmiddleware, wobei die zur Verfügung gestellten Modellierungsklassen syntaktische und sonstige Besonderheiten der Programmiersprache C++ ausnutzen, mit dem Ziel einer möglichst intuitiven, fehlerresistenten und kompakten Modellgeneratorenerstellung.
Die zuvor beschriebenen gemeinsamen, internen Modellierungsklassen enthalten bereits den Großteil der Funktionalität der C++-Endnutzerbibliothek, so dass um diese internen Klassen lediglich ein dünner Wrapper gelegt werden braucht und eventuell. einige zusätzliche Funktionen zur Erhöhung der Usability hinzugefügt werden können. Das Konzept sieht vor, dass die Klassen des C++-Endnutzerschnittstellenmoduls die gleichen Namen wie die internen Modellierungsklassen tragen und von diesen internen Klassen abgeleitet sind, so dass alle öffentlichen Methoden der internen Klassen auch dem Benutzer des C++-Moduls zur Verfügung stehen. Getrennt sind interne und externe Klassen nur durch die unterschiedlichen Namespaces `ModelerIntern` und `Modeler`, somit ist `ModelerIntern::CModel` die interne und `Modeler::CModel` die externe Klasse. Die Trennung von internen und externen Klassen bei sehr ähnlicher Funktionalität dient vor allem dazu, die internen Interfaces änderbar zu halten, was bei publizierten Interfaces in der Regel nicht ohne clientseitige Anpassungen möglich ist.

6.4.2.1 Modellierung mit überladenen Operatoren

Wichtigstes Merkmal der C++-Klassenbibliothek sind überladene Operatoren zur natürlichen Formulierung von Zielfunktion und Restriktionen. Ziel ist dabei, innerhalb des Modellgeneratorcodes mit Modellierungsobjekten Ausdrücke formen zu können, die denen einer Modellierungssprache ähneln. C++ erlaubt die Redefinition bzw. Überladung von fast allen Operato-

ren, beispielsweise von +, -, *, /, +=, -=, =, ==, >= und <=. Ausgenommen sind lediglich einige hier irrelevante Operatoren wie :: oder #. Es werden *binäre Operatoren* unterschieden, die zwei Argumente verknüpfen (z.B. Multiplikationszeichen *) sowie *unäre Operatoren*, die sich nur auf ein einzelnes Argument auswirken (z.B. Minuszeichen zur Kennzeichnung einer negativen Zahl). Die Redefinition eines Operators ist in C++ immer an eine Funktion gekoppelt, die den Transformationsalgorithmus enthält und den Ergebnistyp des Operators bestimmt. Diese Funktion kann entweder eine Methode einer der am Operatorenausdruck beteiligten Klassen sein oder eine globale, klassenunabhängige Funktion.

Tabelle 18 listet die überladenen Operatoren mit ihren Eingangs- und Ergebnistypen für die Grundbausteine Variablen, Koeffizienten, Terme, lineare Ausdrücke und Restriktionen auf. Zunächst gibt es die Operatoren + und -, deren Ergebnis immer ein Objekt vom Typ *Linearer Ausdruck* (`CLinEx`) ist. Ein Term (`CTerm`) ist das Ergebnis der multiplikativen Verknüpfung einer Konstanten (double, int) mit einer Variablen (`CVar`) oder einem Term. Der Divisionsoperator entspricht der Multiplikation mit dem Kehrwert einer Konstanten und ein unäres Minuszeichen kommt der Multiplikation einer Variablen oder eines Terms mit -1 gleich und resultiert in einem `CTerm`-Objekt. Abgesehen von den Vergleichsoperatoren <=, >=, ==, haben alle anderen Verknüpfungen ein Objekt vom Typ `CLinEx` zum Ergebnis. Lineare Ausdrücke können ihrerseits beliebig verkettet oder mit Konstanten multipliziert werden, indem jeder einzelne interne Term des linearen Ausdrucks mit der Konstanten multipliziert wird. Die auf `CLinEx`- und `CRes`-Objekte angewendeten C++-typischen Operatoren += und -= dienen dem Hinzufügen oder Entfernen einer oder mehrerer Variablen. Die Vergleichsoperatoren <=, >= und == haben als Ergebnis ein Restriktions-Objekt, dabei muss aber keineswegs Normalform (Variablen auf der linken, Konstante auf der rechten Seite) eingehalten werden, es können also auch beiden Seiten des Vergleichsoperators Ausdrücke mit Variablen enthalten sein. Die Normalformerzeugung erfolgt intern für den Benutzer verborgen.

Operator	Rechte Seite	Linke Seite	Ergebnis
+	CVar	CVar	CLinEx
+	CVar	CVarSet	CLinEx
+	CVar	double, int	CLinEx
+	CVar	CTerm	CLinEx
+	CVar	CLinEx	CLinEx
+	CVarSet	wie bei CVar	CLinEx
+	CTerm	wie bei CVar	CLinEx
+	CLinEx	wie bei CVar	CLinEx
+	double	CVar	CLinEx
+	double	CVarSet	CLinEx
+	double	CTerm	CLinEx
+	double	CLinEx	CLinEx
– (binär)	Ebenso wie +		

unär –	CVar	-	CTerm
unär –	CVarSet	-	CLinEx
unär –	CTerm	-	CTerm
unär –	CLinEx	-	CLinEx
*	CVar	double	CTerm
*	CVarSet	double	CLinEx
*	CTerm	double	CTerm
*	CLinEx	double	CLinEx
*	double	CVar	CTerm
*	double	CVarSet	CTerm
*	double	CTerm	CTerm
*	double	CLinEx	CLinEx
/	CVar	double	CTerm
/	CVarSet	double	CLinEx
/	CTerm	double	CTerm
/	CLinEx	double	CLinEx
+=	CLinEx	double	CLinEx
+=	CLinEx	CVar	CLinEx
+=	CLinEx	CVarSet	CLinEx
+=	CLinEx	CTerm	CLinEx
+=	CLinEx	CLinEx	CLinEx
+=	CRes	CVar	CRes
+=	CRes	CVarSet	CRes
+=	CRes	CTerm	CRes
+=	CRes	CLinEx	CRes
-=	Ebenso wie +=		
<=	CVar	double	CRes
<=	CVar	CVar	CRes
<=	CVar	CVarSet	CRes
<=	CVar	CTerm	CRes
<=	CVar	CLinEx	CRes
<=	CVarSet	wie bei CVar	CRes
<=	CTerm	wie bei CVar	CRes
<=	CLinEx	wie bei CVar	CRes
<=	CRes	double	CRes
>=	Ebenso wie <=		
==	Ebenso wie <=		

Tabelle 18: Operatorüberladungen

Weiterhin sind noch Überladungen des =-Operators erforderlich, die das Kopieren etwa zwischen zwei `CVar`- oder `CRes`-Objekten ermöglichen und dabei die internen Referenzen aktualisieren. `CResSet` ist zwar in der Tabelle nicht aufgeführt, sollte aber ähnlich wie `CRes` überladen werden, so dass Restriktionsausdrücke formuliert werden können, die eine ganze Gruppe von Restriktionen zugleich betreffen. Ferner wird der Operator [] für den Zugriff auf Indexsets und eindimensionale DataArrays, VarSets und ResSets überladen.

Bei einer realen Implementation der C++-Schnittstellenbibliothek, die im Rahmen dieser Arbeit allerdings nicht durchgeführt wurde, muss auf eine effiziente Programmierung der Überlademethoden geachtet werden, da beim (unnötigen) Erzeugen von Modellierungsobjekten schnell performancegefährdender Overhead entsteht. Nach Möglichkeit muss daher mit Pointern und nicht mit Objektkopien gearbeitet werden, jedoch ohne dass der Benutzer mit der

wenig intuitiven C++-Pointersyntax belastet wird. Hierzu kann in vielen Fällen der C++-Referenzoperator & verwendet werden, der intern eine Referenz bzw. einen Pointer erzeugt, syntaktisch sich aber wie ein Objekt verhält. So sollte beispielsweise eine Überladung des +-Operators für `CVar` deklariert werden mit:

```
CLinEx CVar::operator+(const CVar& v);
```

Um das Thema Operatorüberladung abzuschließen, hier noch einige Syntaxbeispiele, die zeigen sollen, wie mit Modellierungsobjekten und den aufgeführten Operatordefinitionen gearbeitet werden kann:

```
1: CModel m;
2: CSolver s(Solvertype_MOPS);
3: CVar x(m), y(m);
4: x.LB = 123;
5: x + y <= 456;
6: m.AddObj(x + y);
7: s.Optimize(m);
```

Code 1: C++ Codebeispiel Operatorüberladung

In diesem Beispiel werden ein Modell *m* und ein Solver *s* instanziert, und es werden zwei Variablen angelegt, die über ihre Konstruktoren diesem Modell zugeordnet werden. Die Untergrenze von x wird auf 123 gesetzt, und y bleibt auf Defaults (LB=0, UB=Inf). Anschließend wird die erste Restriktion erzeugt: Auf der linken Seite entsteht durch die Verknüpfung zweier `CVar`-Objekte ein implizites Objekt vom Typ `CLinEx`. Der überladene <= Operator bildet aus diesem Objekt und der Konstanten auf der rechten Seite eine Restriktion in Form eines impliziten `CRes`-Objekts, das – für den Benutzer transparent – dem Modell *m* hinzugefügt wird, mit dem die beteiligten Variablen verbunden sind. Die Zielfunktion wird ebenfalls über einen Ausdruck mit überladenen Operatoren gesetzt: x + y ergibt wieder ein implizites `CLinEx`-Objekt, das über die `AddObj`-Methode der Zielfunktion der Modellinstanz hinzugefügt wird. Schließlich wird die Optimierung gestartet, indem die Modellinstanz *m* der Solverinstanz *s* übergeben wird.

```
1: CModel m;
2: CVar x(m), y(m);
3: CLinEx cost;
4: cost = 3*x + y;
5: m.AddObj(cost);
```

Code 2: C++ Codebeispiel Operatorüberladung

Im nächsten Beispiel (Code 2) wird in Zeile 4 die rechte Seite wie folgt ausgewertet: `3*x` ist die Verknüpfung einer Integer-Konstanten mit einem `CVar`-Objekt, was ein implizites `CTerm`-Objekt zum Ergebnis hat. Diesem Term wird wiederum ein `CVar`-Objekt hinzuad-

diert, was in einem `CLinEx`-Objekt resultiert, das schließlich durch die Zuweisung in das `CLinEx`-Objekt `cost` kopiert wird und dann in Zeile 5 der Zielfunktion hinzugefügt wird.

```
1: CModel m;
2: CVarSet x(m, CIndexSet<int> (2006,2007,2008));
3: CRes r(m);
4: x.SetLB(10);
5: r <= 123;
6: for(x.MoveFirst(); !x.IsBeyondLast(); x.MoveNext)
7:     r += x.GetCurrentVar()
```

Code 3: C++ Codebeispiel Operatorüberladung

```
1: CModel m;
2: CVarSet x(m, CIndexSet<int> (2006,2007,2008));
3: CRes r(m);
4: x.SetLB(10);
5: r = x[2006] + x[2007] + x[2008] <= 123;
```

Code 4: C++ Codebeispiel Operatorüberladung

Die nächsten beiden Beispiele (Code 3 und Code 4) tun das Gleiche auf unterschiedliche Art: Beide legen ein einfach indiziertes Variablenset *x* an, das zum Modell *m* gehört und drei Variablen mit den ganzzahlig-numerischen Indizes *2006, 2007* und *2008* enthält. Anschließend wird eine Restriktion r angelegt und in den Zeilen 4 die Untergrenze für alle Variablen des Sets auf zehn gesetzt. Im Beispiel von Code 3 wird in den Zeilen 6 und 7 die Verwendung des Set-Iterators in einer `for`-Schleife demonstriert. Es wird über alle Variablen des Sets iteriert, bis man hinter der letzten angelangt ist, und bei jedem Durchlauf der Schleife wird die aktuelle Variable der Restriktion *r* hinzugefügt. Das Beispiel in Code 4 hingegen zeigt die Verwendung des überladenen `[]`-Operators, der zum Zugriff auf indizierte Sets benutzt werden kann. Der mittlere Ausdruck in Zeile 5 ergibt ein implizites `CLinEx`-Objekt, das zusammen mit der RHS-Konstante in die Überladung des Operators <= eingeht, der eine Restriktion erzeugt, die *r* zugewiesen wird.

```
1: CModel m;
2: CVarSet x(m);
3: x.Name = "X";
4: CMultiIndexSet mis(2);
5: mis.SetIndexSet(1, CIndexSet<int> (1,2,3));
6: mis.SetIndexSet(2, CIndexSet<CString> ("A","B","C"));
7: x.SetIndexSets(mis);
8: x.SetType(MOPS_VarType_Binary);
9: x == 1;
```

Code 5: C++ Codebeispiel Operatorüberladung

Code 5 zeigt die Verwendung der Klasse `CMultiIndexSet`: Zunächst wird ein zum Model *m* gehöriges Variablenset angelegt und ihm in Zeile 3 der Name „X“ gegeben. In Zeile 4 wird ein zweidimensionales `MultiIndexSet` instanziert und in den darauf folgenden beiden

Zeilen wird der erste Index als Zahlen- und der zweite als String-Index mit jeweils drei Werten initialisiert, so dass das Multiindexset die Tupelfolge (1,A), (1,B) …(3,C) repräsentiert. Zeile 7 initialisiert das `CVarSet`-Objekt *x* mit dem MultiIndexset, wobei im Hintergrund die entsprechenden neun Variablen *X.1.A, X.1.B, …, X.3.C* im Modell erzeugt werden. Zeile 8 bestimmt, dass alle Variablen des Sets Binärvariablen sind. Die letzte Zeile erzeugt eine namenlose Restriktion für das Modell *m*, die festlegt, dass die Summe aller neun Variablen des Variablensets *x* gleich 1 ist. Das Beispiel zeigt, dass die Verwendung der Modellierungsklassen oftmals eigene Schleifenkonstrukte überflüssig macht und so zu einem kompakteren und tendenziell auch weniger fehleranfälligen Code führt.

6.4.2.2 Ausnahmebehandlung

Die Methoden der prozeduralen Schnittstellen sind durchweg nach dem Return-Code-Prinzip entworfen, d.h. eine Methode liefert einen Rückgabewert vom Typ `Integer`, der Aufschluss über Erfolg oder Misserfolg der Methode gibt. Effizienter und eleganter ist jedoch die objektorientierte Methode der strukturierten Ausnahmebehandlung. Dabei löst eine Methode im Fehlerfall eine Exception aus, die innerhalb eines `try{…}`-Blocks aufgefangen wird und im optional folgenden `catch{…}`-Block behandelt wird. Vorteil ist, dass nicht nach jedem Methodenaufruf ein Return Code angefragt werden muss. Wenn kein Return Code zurückgegeben werden muss, so eröffnet dies die Möglichkeit, andere Werte oder Objekte zurückzugeben, was gerade bei diversen `Get…()`-Methoden zu einer intuitiveren Syntax führt. So kann beispielsweise aus der Return-Code-Variante

```
rc = myDataArray.GetElementAt(&myElement, Index1, Index2);
```

bei Einsatz strukturierter Ausnahmebehandlung formuliert werden

```
myElement = myDataArray.GetElementAt(Index1, Index2);
```

Die zweite Version der Methode muss mit dem Zusatz `throw(…)` deklariert werden, um deutlich zu machen, dass sie eine Exception auslösen kann.

6.4.3 COM-Komponentenbibliothek

Die COM-Komponentenbibliothek ist Teil des Middlewarekonzepts und stellt neben *MOPS Studi*o den Schwerpunkt der hier geleisteten Implementationsarbeit dar. Sie ist gleichzeitig aufgrund des Nichtvorhandenseins in den meisten anderen Solver-Schnittstellensystemen ein Alleinstellungsmerkmal. Die konzeptionellen Aspekte einer solchen Bibliothek werden in den folgenden Abschnitten beschrieben. Details der Implementation – die aus Aufwandsgründen teilweise vom hier beschriebenen Entwurf abweicht – finden sich in Kapitel 7.2.

6.4.3.1 Überblick

Der wichtigste Unterscheid zur zuvor beschriebenen C++-Modellierungsbibliothek ist, dass es sich bei einer COM-Bibliothek um eine Sammlung von Binärkomponenten handelt, die ohne vorherige Kompilation unmittelbar in einer Vielzahl unterschiedlicher Programmierumgebungen verwendbar ist. Dies impliziert, dass es keine universelle Syntax für die Ansteuerung von COM-Komponenten gibt, die zur Erreichung einer möglichst intuitiven Modellgeneratorprogrammierung ausgenutzt werden könnte, wie dies etwa bei der C++-Bibliothek der Fall ist. Die Einbindung von COM-Komponenten erfolgt immer sprachspezifisch entsprechend der Clientumgebung. Daher sollte das Design einer COM-Bibliothek auf eine bestimmte Clientprogrammiersprache abzielen, wenn deren syntaktische und sonstige Eigenheiten optimal unterstützt werden sollen. Hier wurde als primäre Zielsprache Visual Basic (VB6, VBA, VB.NET) gewählt, da diese COM in umfassender Weise unterstützt. Zwar ist VB6 inzwischen von VB.NET abgelöst, aber bei Altapplikationen und vor allem beim Scripting für Microsoft Office (VBA) hat die Sprache immer noch große Bedeutung. Außerdem sind COM-Modellierungskomponenten dank der COM/NET-Interoperabilitätsmechanismen (vgl. Abschnitt 4.3.3) auch in den .NET-Sprachen einsetzbar. Die relativ langsame Interpreter-Sprache VB6 profitiert ferner von Performance-Vorteilen, die durch die Verschiebung von Modellgenerierungsaufgaben in die COM-Komponenten resultieren (z.B. Vermeidung von Schleifen). Einige Designentscheidungen wurden bewusst so getroffen, dass die Komponenten OLE-automatisierungskonform sind und Visual Basic in optimaler Weise unterstützt wird. In anderen Programmiersprachen, vor allem in C++, sind die so konzipierten COM-Schnittstellen jedoch oft umständlich und wenig intuitiv nutzbar (etwa bei Verwendung von SafeArrays). Dies ist aber nicht als Nachteil anzusehen, denn für C++ ist ohnehin die spezifische und besser angepasste C++-Modellierungsbibliothek vorgesehen.

Die Komponenten der COM-Bibliothek spiegeln in großen Zügen das Design der gemeinsamen inneren Modellierungsklassen wider, auf denen sie beruhen, denn konzeptgemäß benutzen die COM-Komponenten zur internen Implementation ihrer Features die gemeinsamen C++-Klassen (Abschnitt 6.4.1)[1]. Abbildung 26 zeigt die Komponenten der COM-Bibliothek, wobei die Ähnlichkeit mit Abbildung 25 aus dem gemeinsamen Designansatz resultiert.

[1] Aus Gründen des Implementationsaufwands wurde dies in der prototypischem Umsetzung der COM-Bibliothek jedoch so nicht umsetzt. Die COM-Bibliothek implementiert daher sämtliche Features zunächst direkt, ohne interne, gemeinsame C++-Klassen. Details siehe Kapitel 7.2.

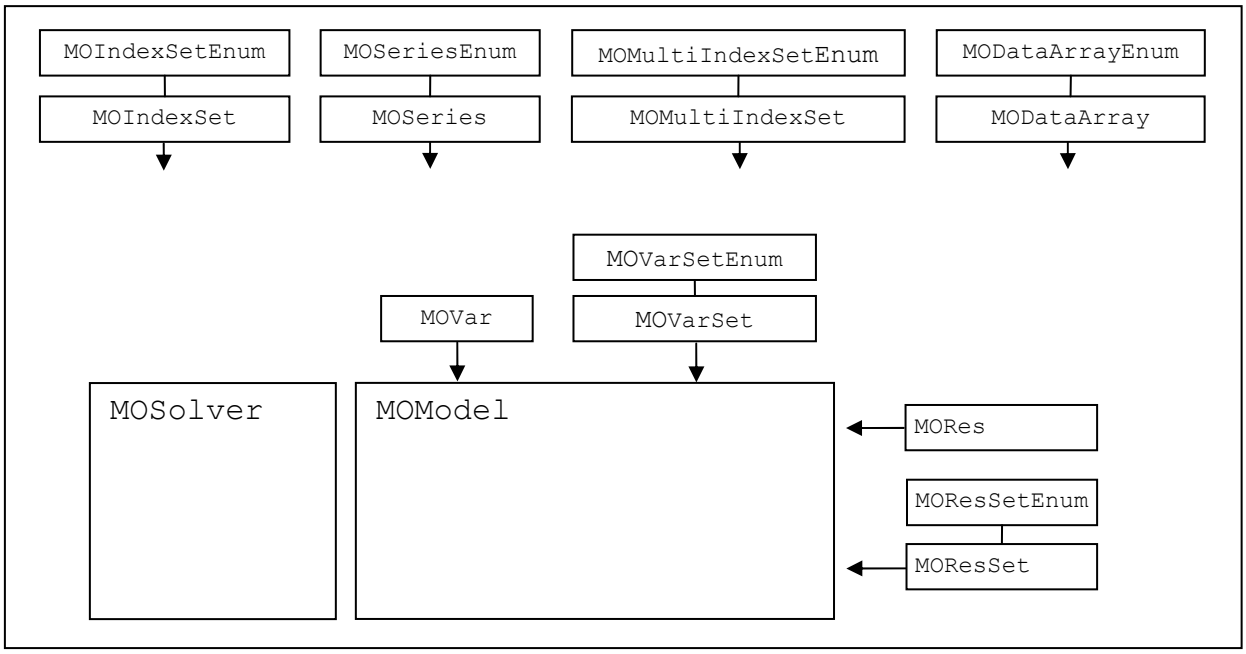

Abbildung 26: COM-Bibliothek

Das Präfix „MO“ der Komponentennamen soll die Zugehörigkeit zur Modeler-Bibliothek deutlich machen. Im Zentrum steht die Komponente `MOModel`, die eine Instanz des Modells enthält und die – genauso wie die Klasse `CModel` der C++-Bibliothek – alle Methoden und Eigenschaften enthält, die für matrix- und objektorientierte Modellierung notwendig sind. `MOModel` übergibt das Modell zur Lösung an eine Instanz von `MOSolver`, die einen Solver kapselt und diesen über den Solver-Manager steuert. Die beiden Komponenten sind für einfache Matrixgenerierung Laden und Speichern von Modellen, Optimierung und Lösungsabfrage ausreichend. Alle weiteren Komponenten stellen zusätzliche Modellierungsobjekte dar, die in ihren Grundfunktionalitäten teilweise bereits in Abschnitten 6.4.1 und 6.4.2 erläutert wurden. Hinzugekommen sind die *x*`Enum`-Komponenten, die in COM-typischer Weise eine Iteration bzw. Enumeration von Collections erlauben. Ebenfalls spezifisch für die COM-Bibliothek ist `MOSeries`, eine Komponente für einfache Indexsets, die auf einer arithmetischen Reihe beruhen. Zu den C++-Klassen `CTerm` und `CLinEx` analoge Komponenten sind nicht vorgesehen, da die Verknüpfung von Variablen, Termen und linearen Ausdrücken mangels Operatorüberladung in COM/VB nicht funktioniert und somit diese Klassen ihres Hauptzweckes beraubt sind. Alle Komponenten sind in der COM-Bibliothek *Modeler COM Library* vereinigt, die in einer prozessinternen Version als DLL und in der prozessexternen Version als EXE distribuiert wird.

6.4.3.2 Detailaspekte

Die allgemeinen COM- und VB-Spezifikationen implizieren einige Besonderheiten in Bezug auf das Design des COM-Schnittstellenmoduls. Im Detail sollen daher folgende Punkte behandelt werden:

- Prozessinterne und -externe Komponenten sowie DCOM
- Aggregation
- Fehlende Methodenüberladung und `Variant`-Typen
- `SafeArrays` und variable Argumentlisten
- Auswirkungen fehlender Operatorüberladung
- Enumeratoren
- Methoden und Eigenschaften

Prozessinterne und -externe Komponenten sowie DCOM

Wie bereits in 4.3.3 dargelegt, bietet das Component Object Model Ortstransparenz in Bezug auf den Ausführungsort des Komponentencodes. So ist es für einen Client gleichgültig, ob eine Komponente in seinem Prozessraum, in einem anderen Prozessraum oder auf einem anderen Rechner ausgeführt wird. Diese Tatsache wird in der COM-Komponentenbibliothek ausgenutzt, indem der selbe Codekern zur Erzeugung entsprechend unterschiedlicher Binärversionen genutzt wird. Prozessinterne und -externe sowie Remotekomponenten unterscheiden sich nämlich nicht im Sourcecode der Methoden und Eigenschaften, sondern nur in Bezug auf die Proxy- und Stub-Routinen für das prozess- oder rechnerübergreifende Marshalling. Da diese vom MIDL-Compiler automatisch erzeugt werden, muss der Entwickler lediglich unterschiedliche Projekte zur Erzeugung mit einigen bedingten Compilerdirektiven einrichten, um die verschiedenen Binärversionen (DLL oder EXE) zu erzeugen. Die Verfügbarkeit unterschiedlicher Komponentenversionen ist ein bedeutsamer Vorteil, da er einen situativ angepassten Einsatz des Optimierers und eine bessere Performance-Skalierung ermöglicht. So kann beispielsweise eine prozessexterne COM-Komponente in Form einer EXE unter Windows32 über den gesamten virtuellen Prozessraum von 2 GB (3 GB mit Switch */3GB*) verfügen, während eine prozessinterne Komponente in einem von Code und Daten der Applikation teilweise belegten und fraktionierten Prozessraum arbeiten muss. Daher kann gerade für den in der Implementation verwendeten Solver MOPS die Fraktionierung des Prozessraums problematisch sein, da MOPS zusammenhängenden Speicher (internes `b`-Array) benötigt. Bei manchen System- und Applikationskonfigurationen konnte beobachtet werden, dass bei-

spielsweise trotz einer Belegung des 2 GB großen Speicherraums mit lediglich 100 MB, der größtmögliche zusammenhängende Speicherblock durch Fraktionierung bei ca. 1.200 MB liegt, was die Maximalgröße des Optimierungsmodells empfindlich beschränkt. Eine prozessexterne Komponente besitzt dagegen einen zumindest theoretisch freien 2 bzw. 3 GB großen Prozessraum. In der Praxis kann es allerdings auch hier durch bestimmte Systemprogramme wie Antivirustools zur Fraktionierung kommen. Neben einem größeren unfraktionierten Speicherraum haben gesonderte Prozesse außerdem eine bessere Performanceskalierung, so dass vor allem auf Mehrprozessorsystemen die gleichzeitige Lösung mehrerer Modelle ermöglicht wird, was beispielsweise mit der ursprünglichen MOPS.DLL nicht möglich ist. Dies gilt ebenso für die Remote-Version (DCOM) der Komponentenbibliothek, die die parallele Optimierung auf verteilten Rechnern ermöglicht. Nachteil bei Prozess- oder rechnerexternen Komponenten ist allerdings, dass alle Kommunikation (Methodenaufrufe, Datenübergabe) mit Marshalling bewältigt werden muss, was einen erheblichen Overhead und entsprechende Performanceeinbußen impliziert.

Aggregation

In Analogie zum Vererbungsverhältnis zwischen `CModel` und `CMatrix` bei den internen C++-Klassen wäre denkbar, ein solches Design auch für die Komponentenbibliothek zu wählen. In COM stehen die (Pseudo-)Vererbungsmechanismen *Containment* und *Aggregation* zur Verfügung (vgl. [Willliams94]), wobei hier eine Aggregation eines inneren `MOPSMatrix`-Interfaces zum äußeren `MOPSModel`-Interface in Frage käme. Für einen Client der äußeren Komponente erscheinen dann die Eigenschaften und Methoden der inneren als natürlicher Bestandteil der äußeren Komponente. Anders als in C++, wo es z.B. über `friend`-Verhältnisse für die Child-Klasse Möglichkeiten gibt, auf Datenstrukturen der Parent-Klasse zuzugreifen, bleiben bei COM die Interna der inneren Klasse in Binärform vom Zugriff durch die äußere Klasse geschützt, so dass jegliche Kommunikation über das Interface der internen Komponenten erfolgen muss. Da die Aufspaltung in zwei aggregierte Komponenten außer einer logischen Trennung von matrix- und modellierungsobjektbezogenen Methoden nur zusätzlichen Overhead der COM-Interfacekommunikation bedeutet, wird auf ein derartiges Design verzichtet, so dass es lediglich eine einzige Komponente `MOModel` gibt, die die interne Klasse `CModel` zur Implementation ihrer Funktionalitäten nutzt.

Fehlende Methodenüberladung und `Variant`-Typen

Eine weit verbreitete Technik in Sprachen wie C++, Java, C# oder VB.NET ist die Überladung von Funktionen oder Methoden. Dabei werden mehrere Versionen einer Funktion oder Methode definiert, die zwar den gleichen Namen haben, aber sich in Zahl oder Art ihrer Parameter unterscheiden und daher unterschiedliche Signaturen besitzen. Während der Übersetzung weist der Compiler einem Funktions- oder Methodenaufruf dann anhand der Argumente die jeweils passende Version zu. Die gemeinsamen C++-Modellierungsklassen und die C++- und .NET-Endnutzermodule machen zum Wohle einer höheren Programmierintuitivität ausgiebigen Gebrauch von dieser Möglichkeit. Leider unterstützen weder COM noch VB6/VBA Methodenüberladung. Als syntaktische Alternative bietet sich die Verwendung von `Variant`-Typen an, die in VB6/VBA in sehr benutzungstransparenter Weise eingesetzt werden können.

Beim `Variant`-Datentyp handelt es sich intern um eine Struktur mit inneren `Union`-Elementen, die in der Lage ist, eine ganze Reihe unterschiedlicher Datentypen zu speichern. Dabei kann es sich um skalare Typen wie `Double` oder `Long`, aber auch um Pointer auf Strings, SafeArrays oder auf Interfaces von COM-Objekten handeln. VB macht ausgiebigen Gebrauch von Variants und nimmt an den meisten Stellen Typumwandlungen automatisch vor. Diese Tatsache kann ausgenutzt werden, um trotz fehlender Überladbarkeit Methodenaufrufe variabel bezüglich ihrer Argumenttypen zu machen. Dies sei anhand eines einfachen Beispiels erläutert: Die Klasse `CModel` der C++-Modellierungsbibliothek hat eine überladene Methode `DelCol` zum Löschen einer Spalte, die sowohl mit einem Integer-Wert für den Spaltenindex als auch mit einem String für den Spaltennamen aufgerufen werden kann, deklariert als

```
int DelCol(int iColIndex);
int DelCol(char* sColName);
```

Für die COM-Komponente wird zum gleichen Zweck in *Interface Definition Language* (MIDL) deklariert

```
HRESULT DelCol([in]VARIANT Col);
```

Ein Aufruf dieser Methode kann in VB6/VBA und (dank der Interoperabilitätsmechanismen) ebenfalls in VB.NET wahlweise erfolgen mit

```
myMOPSModel.DelCol (123)      oder
myMOPSModel.DelCol ("abc")
```

Visual Basic wandelt dabei beide Argumente implizit in Variants um und setzt den Variant-Subtyp (intern das Element `VARTYPE` der `VARIANT`-Struktur) auf Long bzw. String (genauer gesagt BSTR). Es existiert nur eine einzige Version der Methode, in der zur Laufzeit überprüft werden muss, um welchen Variant-Subtypen es sich handelt. Je nach Typ müssen unterschiedliche Verarbeitungswege eingeschlagen werden, oder es wird ein Fehler ausgelöst, falls ein anderer, nicht unterstützter Typ angegeben wurde. Diese „Pseudo-Überladung" funktioniert zwar zunächst nur bei Methoden mit gleicher Argumentzahl, kann aber durch Verwendung optionaler Argumente auch auf Methoden mit unterschiedlicher Argumentzahl ausgedehnt werden – was in der COM-Bibliothek auch an verschiedenen Stellen geschieht.
In Bezug auf `Variant`-Typen sei noch erwähnt, dass diese auch für optionale Argumente in Funktions- und Methodenaufrufen verwendet werden können. Dazu müssen die jeweiligen Argumente mit dem MIDL-Attribut optional gekennzeichnet werden, wie z.B.:

```
HRESULT myMethod([in, optional] VARIANT myArgument);
```

Wurde das entsprechende Argument ausgelassen, so übergeben Programmiersprachen, die optionale COM-Argumente unterstützen, einen impliziten Variant vom Typ `VT_ERROR` mit dem Scode-Wert `DISP_E_PARAMNOTFOUND`. Die Routine muss dies abprüfen und einen geeigneten Defaultwert annehmen. Auch diese Technik wird an vielen Stellen in der COM-Optimierungsbibliothek eingesetzt.

SafeArrays und variable Argumentlisten

`SafeArray` ist ein in COM/OLE weit verbreiteter Datentyp zur Abbildung ein- und mehrdimensionaler Felder, deren Typ aus einer Untermenge der Variant-Typen sein muss. Somit kann ein `SafeArray` Typen wie `Long`, `Double` oder `BSTR` enthalten. Erlaubt sind auch Variant-Pointer, über die indirekt sämtliche Variant-Typen abbildbar sind. COM-Komponenten können in Visual Basic mithilfe von SafeArrays-Argumentlisten mit variabler Anzahl an Methodenargumenten implementieren. Solche Methoden werden in MIDL deklariert mit

```
[… vararg] HRESULT myMethod([in]SAFEARRAY(VARIANT)* myList);
```

Der Methode wird dann ein Pointer auf ein eindimensionales, aus Variants bestehendes SafeArray übergeben, dessen Elemente unterschiedliche Datentypen wie z.B. Long, Double, BSTR, oder COM-Interfacepointer (`IDispatch`) sein können. Die so übergebenen Argu-

mentlisten mit variabler Elementzahl, müssten dann innerhalb der Methode auf Typ- und Wert-Validität überprüft werden. Diese Technik der Übergabe von Argumentlisten wird an mehreren Stellen in der MOPS COM-Bibliothek eingesetzt. Sie unterscheidet sich erheblich von dem in C/C++ üblichen Verfahren (Makros `va_arg`, `va_end`, `va_start`).

Auswirkungen fehlender Operatorüberladung

Die Überladung von Operatoren, die in der C++-Bibliothek das Kernstück der Modellgenerierung mit Optimierungsobjekten bildet, ist mit COM-Komponenten nicht möglich. Auch dann nicht, wenn die Komponenten in eine Programmiersprache wie C# eingebunden werden, die prinzipiell Operatorenüberladung kennt. Dies hat gravierende Auswirkungen auf das Design der COM-Bibliothek: Eine Modellformulierung wie in den C++-Beispielen in Abschnitt 6.4.2 ist somit nicht möglich. Daher machen zu den Klassen `CTerm` und `CLinEx` analoge COM-Komponenten zur Unterstützung der Restriktionsformulierung keinen Sinn. Die Komponentenäquivalente anderer interner Klassen, wie `CRes` oder `CVar` können hingegen sehr wohl sinnvoll zur vereinfachten Modellgenerierung eingesetzt werden, selbst wenn keine Ausdrücke mit überladenen Operatoren formuliert werden können.

Enumeratoren

Als *Iteratoren* oder *Enumeratoren* werden Klassen oder Komponenten bezeichnet, mit denen eine geordnete Menge von Elementen durchlaufen werden kann. Beide Begriffe werden hier synonym benutzt, wobei man im Allgemeinen von *Iteratoren* spricht (etwa [Gamma95], [Booch94]). Im Zusammenhang mit COM/OLE wird jedoch meist der Begriff *Enumerator* benutzt, da auch die entsprechenden Interfaces so heißen (`IEnum...`). [Gamma95] beschreibt Iteratoren als abstraktes Entwurfsmuster, das in vielfältigen Zusammenhängen Einsatz finden kann und nennt die notwendigen Grundfunktionen eines Iterators `First()`, `Next()`, `IsDone()` und `CurrentItem()`. Die COM-Bibliothek besitzt für alle Komponenten, die als geordnete Mengen aufgefasst werden können, gesonderte Iterator- bzw. Enumerator-Komponenten (z.B. `MOIndexSetEnum`, `MOVarSetEnum` etc.). Zudem verfügen die Set-Komponenten selbst auch über integrierte Enumeratoren, so dass ein Set sowohl von sich selbst, als auch von einer Enumerator-Komponente durchlaufen werden kann. Zudem enthalten die Set-Komponenten Factory-Methoden (zum Factory-Schema vgl. [Gamma95]) zur Erzeugung neuer Enumerator-Objekte. Der Benutzer kann Filter für Enumerator-Komponenten setzten, so dass diese nicht alle, sondern nur eine Teilmenge der Elemente der Basismenge

durchlaufen, was in Modellgeneratoren häufig genutzt werden kann. Die Hauptaufgabe der Enumeratoren ist die Bereitstellung des COM-Interfaces `IEnumVARIANT`, das in Visual Basic in `For…Each`-Schleifenkonstrukten genutzt wird. Damit werden syntaktisch sehr intuitive Schleifen über Optimierungsobjekte möglich, etwa über alle Variablenobjekte einer indizierten Variablenmenge:

```
Dim vs As New MOPSVarSet, v As MOPSVar
…
For Each v In vs
…
Next
```

Methoden und Eigenschaften

COM-Komponenten haben aus Benutzersicht in VB Methoden und Eigenschaften. Intern, d.h. auf der Ebene der Interfacespezifikationen und deren Implementationen, existieren jedoch nur Methodenaufrufe. Anders als eine C++-Klasse, die beliebige Datenelemente durch `public`-Deklaration zu öffentlich zugänglichen Eigenschaften machen kann, erfolgt der Zugriff in COM immer über explizite `get_`-Methoden zum Lesen und `set_`-Methoden zum Schreiben. Die „Illusion“ einer vorhandenen Eigenschaft wird dabei von der Client-Programmierumgebung der Komponente erzeugt. VB als Zielprogrammiersprache erlaubt darüber hinaus die Parametrisierung von Eigenschaften, was im Code folgendermaßen aussieht:

```
myValue = myComObject(myParameter1, …)
myComObject(myParameter1, …) = myValue
```

Als Parameter können Listen mit einer festen Anzahl an obligatorischen oder optionalen Argumenten festgelegt werden. Parameterlisten können aber nicht – und das ist eine wichtige syntaktische Einschränkung für das Bibliotheksdesign – eine variable Anzahl an Argumenten haben. In der COM-Bibliothek wird diese Einschränkung relevant beim Zugriff auf Eigenschaften, die mit Indizes parametrisiert sind. Die Bibliothek kann zwar Mengen (`MOVarSet`, `MOResSet`, `MODataArray`) mit beliebig vielen Indexsets versehen, so dass n-dimensionale Objekte entstehen, beim parametrisierten Eigenschaftenzugriff wurde die Anzahl der Parameter jedoch auf fünf begrenzt. Diese leider unvermeidbare Einschränkung führt dazu, dass alle Eigenschaftenzugriffe, die der Angabe von Indexwerten bedürfen, in VB nach folgendem Schema aufgebaut sind:

```
myMOObject.Value([Index1],[Index2],[Index3],[Index4],[Index5])
```

wobei [Index*N*] einen optional anzugebenden Index bezeichnet. Die Verwendung von Eigenschaften zum lesenden oder schreibenden Zugriff auf Datenelemente einer Komponente erhöht vor allem in VB die Intuitivität des Modellgeneratorcodes, wie auch die Beispiele in Abschnitt 8.1 noch zeigen sollen. Eine weitere syntaktische Möglichkeit in VB ist, dass der Name einer als Default gekennzeichneten Eigenschaft (DISPID_VALUE) weggelassen werden kann. So könnte z.B. der Wert eines 2-fach, über Jahre und Monatsnamen indizierten MODataArray-Elementes ausgelesen werden mit

```
myValue = myMODataArray(2005, "Januar")
```

Die Verwendung einer expliziten Get-Methode wäre hingegen weniger elegant:

```
myMODataArray.GetValue(myValue, 2005, "Januar")
```

Vor allem, wenn der Rückgabewert in einen Ausdruck einfließen soll, erspart die erste Variante das Anlegen einer zusätzlichen Hilfsvariablen:

```
myResult = myMODataArray(2005, "Januar") * 4711
```

6.4.3.3 Beschreibung der einzelnen COM-Komponenten

Anhang 5 stellt die Schnittstellen der Komponenten der COM-Bibliothek dar. Zu jeder Schnittstelle existiert eine entsprechende Komponente, die die Methoden und Eigenschaften der Schnittstelle implementiert. Da COM interfaceorientiert ist, und da die Funktionalität einer Komponente ausschließlich über deren explizite Schnittstellen zugänglich ist, reicht zur Darstellung der Funktionalität einer Bibliothek die Darstellung ihrer Schnittstellen. Die softwarearchitektonische Struktur der Implementationskomponenten, die darüber hinaus noch diverse OLE-interne Interfaces besitzen, wird in Kapitel 7.2 erläutert. Die Grafik in Anhang 5 zeigt die Vererbungszusammenhänge zwischen den Interfaces, die vor allem die Vererbung von Enumerator-Funktionalitäten und den Zugriff auf Multi-Indexsets betreffen.

MOModel

Im Zentrum steht die Schnittstelle IMOModel, implementiert in der zentralen Komponente MOModel, die das Modell repräsentiert und die alle Methoden und Eigenschaften zur matrixorientierten Modellgenerierung besitzt. Spaltenattribute (LB, UB etc.) und Zeilenattribute (LHS, RHS etc.), einschließlich der Lösungswerte, sind als Eigenschaften der Komponente

angelegt. Die Eigenschaftsparameter sind Variants, die wahlweise einen Spalten- bzw. Zeilennamen oder den entsprechenden numerischen Index enthalten können. Etwa:

```
myMOModel.ColUB(123) = 47.11
myMOModel.ColUB("X.1") = 47.11
```

Man kann aber nicht nur auf einzelne, sondern auch auf mehrere Spalten oder Zeilen gleichzeitig zugreifen. Dazu wird als Eigenschaftsparameter ein `SafeArray`, ein `MOIndexSet`, ein `MODataArray`, oder ein `MOMultiIndexSet` angegeben. So werden der Elementzahl entsprechend mehrere Werte gesetzt oder zurückgegeben und zwar als `SafeArrays`, `MODataArrrays` oder `MOIndexSets`. Beispielsweise:

```
myMOIndexSet.Add "X.1", "X.2", "X.5"
Dim a(1 to 3) As Double
a = myMOModel.ColUB(myMOIndexSet)
```

Diese Art der variant-basierten Mehrdeutigkeit, die inhaltlich der Methodenüberladung mit unterschiedlichen Argumenttypen gleichkommt, wird innerhalb der COM-Bibliothek an sehr vielen Stellen genutzt.

Zur Erzeugung von Modellierungskomponenteninstanzen besitzt `MOModel` so genannte *Object Factories*, die als Eigenschaften von `MOModel` implementiert sind. Mit ihnen können neue Variablen-, Restriktions-, VariablenSet-, und RestriktionsSet-Objekte erzeugt werden. Als Eigenschaftsparameter können optionale Initialisierungswerte mitgegeben werden, wie z.B. die zur Indizierung eines VariablenSets zu verwendenden Index- oder MultiIndexSets. Über die Eigenschaften `Res` und `Var` können an bereits bestehende Zeilen bzw. Spalten Restriktions- und Variablenobjekte gebunden oder neu erzeugt werden. Die Object Factories wurden als Eigenschaften gestaltet, um sie in Zuweisungsoperationen mit "=" verwenden zu können.

```
Dim myVar as MOVar
myVar = myMOModel.NewVar(0,100)
```

fügt beispielsweise dem Modell eine neue Variable mit Untergrenze 0 und Obergrenze 100 hinzu und liefert in `myVar` ein Variablenobjekt, das diese neue Spalte referenziert. Das nächste Beispiel nutzt die Object Factory der `MOModel`-Komponente zur Erzeugung eines indizierten Variablensets (*Sales.Jan, Sales.Feb, Sales.Mrz*):

```
Dim myIndexSet as new MOIndexSet
```

```
Dim myVarSet as MOVarSet
myIndexSet.Add "Jan", "Feb", "Mrz"
myVarSet = myMOModel.NewVarSet("Sales",myIndexSet)
```

Die vom Interface `IMOModel` angebotenen Methoden betreffen vor allem den Bereich „Modellgenerierung und -modifikation“: Methoden wie `AddCol`, `AddRow` und `AddNZ` dienen zur matrixorientierten Modellgenerierung, während `AddVar`, `AddVarSet`, `AddRes` und `AddResSet` die Modellgenerierung mithilfe von Modellierungsobjekten unterstützen. Es gibt dabei immer ein Methodenpaar: Eine Methode zum Hinzufügen, Einfügen oder Löschen einer einzelnen Zeile oder Spalte und eine andere Methode zur gleichzeitigen Behandlung mehrerer Zeilen bzw. Spalten. Einzelne Variablen- oder Restriktionsobjekte können mit `AttachVar` und `AttachRes` einer bestimmten Spalte oder Zeile angehangen werden, um diese fortan zu repräsentieren. Ferner ist das Hinzufügen von Special Ordered Sets vorgesehen, wozu die betreffenden Variablen als Spaltenindizes, Variablennamen oder Variablenobjekte angegeben werden können und ein ebenfalls zu übergebendes Array die Abfolge der Variablen im SOS bestimmt.
Eine Besonderheit, die aus dem Fehlen der Operatorüberladung in COM resultiert, zeigt sich bei der Methode `ObjFuncSum`: In der C++ Bibliothek wäre dank Operatorüberladung ein Ausdruck wie `myModel.AddObj(2*X + 3*Y)` zum Setzen von Zielfunktionskoeffizienten möglich. Da dies in COM/VB nicht erlaubt ist, erwartet die Funktion `ObjFuncSum` der `MOPSModel`-Komponente zur Erzielung der gleichen Funktionalität eine alternierende Reihe von Koeffizienten und Variablen – entweder als explizite Auflistung wie

```
myModel.ObjFuncSum 2,X,3,Y
```

oder als Arrays

```
Dim c(1 To 2) As Double, v(1 To 2) as Object
c(1) = 2 : c(2) = 3
v(1) = X : v(2) = Y      ' X und Y sind zuvor angelege MOVar-Objekte
myModel.ObjFuncSum c,v
```

Weitere Methoden der Hauptkomponenten betreffen die Ein- und Ausgabe von Modellen in verschiedenen Formaten (MPS, LINDO, AMPL, MPL). Ebenso existieren einige Eigenschaften, die die Modellierungssprachenanbindung betreffen (z.B. `LocationOfAMPLEXE`).

MOSolver

Diese Komponente kapselt Solver-Instanzen und enthält Methoden zur Lösungsprozesssteuerung: Neben der normalen Methode zum Start der Optimierung `Optimize` besitzt die Komponente eine Methode `OptimizeAsync`, die einen Optimierungslauf in einem neuen Thread startet und anschließend sofort mit dem weiteren Programmcode fortfährt, so dass der Thread im Hintergrund die Optimierung ausführt. Bei Beendigung des Optimierungslaufs wird ein Event (`OptimizeAsyncFinished`) ausgelöst, der im Modellgenerator dann von einer entsprechenden Event-Handling-Routine aufgefangen werden kann. Ein im Hintergrund laufender Optimierungsthread kann mit `StopOptimizeAsync` beendet werden. Die komplette Threadsteuerung wird so vor dem Benutzer verborgen.

Eine Besonderheit ist die Methode `ShowPropertyPage`(): Wird die Komponente als Control kompiliert, so besitzt sie zur Entwurfszeit eine Eigenschaftsseite (Property Page), die ihrerseits ein eigenständiges Control ist und zur menügesteuerten Eingabe der Solver-Parameter dient. Diese Eigenschaftsseite lässt sich über besagte Methode auch zur Laufzeit anzeigen, so dass Applikationen, die die Komponentenbibliothek benutzen, keine eigenen Menüs für Solver-Parameter bereitstellen müssen. MOPS Studio macht sich beispielsweise diese Technik für den Solver MOPS zu Nutze.

`MOModel` und `MOSolver` sind konzeptionell zwei getrennte Komponenten, die aber aus Gründen der Vereinfachung in der prototypischen Implementation gemeinsam in einer Komponente implementiert wurden, da dort bisher nur ein einziger Solver angebunden wurde.

MOVar und MORes

Diese beiden Komponenten stellen eine Modellvariable bzw. eine Modellrestriktion dar und verhalten sich wie eine logische Referenz in Objektform. Sie werden entweder durch die Object Factories des `MOModel` mit `NewVar` bzw. `NewRes` erzeugt oder autonom instanziert und dann dem Modell hinzugefügt (`AddVar`, `AddCol`) oder aber einer bereits bestehenden Spalte bzw. Zeile referenziell angeheftet (`Attach`, `AttachVar`, `AttachRes`). Über `IndexInModel` lässt sich der absolute Matrixindex im Modell auslesen, auf den die Objekte verweisen. Die beiden Komponenten bieten Zugriff auf alle Variableeln- und Restriktionsattribute (UB, LB etc.) sowie die Lösungswerte (LPSol etc.).

`MORes` erlaubt darüber hinaus mit den Methoden `Add` und `Del` das Hinzufügen von Koeffizienten und Variablen zur Restriktion, wobei `Add` eine beliebig lange, alternierende Liste von Koeffizienten und `MOVar`- oder `MOVarSet`-Objekten erwartet. Beispiel:

```
Dim X as MOVar, Y as MOVar, r as MORes
X = myModel.NewVar : Y = myModel.NewVar
r = myModel.NewRes
r.SetEQ 123
r.Add 2,X, -5,Y
```

Damit wird die Restriktion $2 \cdot X - 5 \cdot Y = 123$ erzeugt. Diese Konstruktion dient als Ersatz für die nicht vorhandene Möglichkeit der Operatorüberladung, die derartige Formulierungen in C++ in eleganterer Form erlaubt.

MOIndexSets

Die Komponente `MOIndexSet` mit ihrem Interface `IMOIndexSet` dient der Speicherung einfacher, geordneter Indexmengen, kann aber auch für andere Arten von Mengen eingesetzt werden. Indexmengen dürfen keine doppelten Elemente enthalten, für andere Arten von Mengen kann dies jedoch über die Eigenschaft `AllowIdenticalElements` erlaubt werden. Neben dem Hinzufügen, Einfügen und Löschen von Elementen werden auch folgende Mengenoperationen unterstützt: Schnittmengen mit der Methode `Intersect`, Vereinigungsmengen mit `Add` und Differenzmengenbildung mit `Del`. Ferner gibt es Rechenmethoden zur Vektoraddition mit anderen IndexSets oder Arrays und zur Ermittlung des Vektorprodukts. Die Enumerator-Factory `Enum`() liefert Enumerator-Komponenten, die optional über einen Teilbereich des Sets mit einer wählbaren Schrittweite iterieren können. Die Eigenschaft Set liefert als Rückgabewert eine Referenz auf sich selbst und initialisiert sich mit den angegebenen Parameterwerten. Dies dient der Erzeugung einer Syntax, die ähnlich der in C++ gebräuchlichen Konstruktor-Initialisierung ist, wie folgendes Beispiel zeigt:
Eine C++-Funktion, die als Argument ein Objekt von Typ `CIndexSet` erwartet, kann aufgerufen werden mit

```
someFunction(CIndexSet(1,2,3))
```

falls `CIndexSet` einen entsprechenden Konstruktor für eine variable Anzahl von Elementen besitzt. Die Eigenschaft Set erlaubt eine ähnliche Syntax in COM/VB, wo es allerdings per Design keine Konstruktoren gibt:

```
Dim s as New MOIndexSet
someFunction(s.Set(1,2,3))
```

MOSeries

Diese Komponente enthält eine arithmetische Reihe und kann wie ein IndexSet zur einfachen Indizierung verwendet werden. Im Unterschied zu `MOIndexSet` werden die Elemente aber intern nicht einzeln gespeichert, sondern nur Startelement, Endelement und Schrittweite. Wie das Interface `IMOIndexSet`, verfügt das Interface der Komponente auch über die von `IEnumVARIANT` und `IMOCustomEnum` geerbten Enumerator-Funktionalitäten.

MOMultiIndexSets

Wie bei den gemeinsamen Modellierungsklassen und der C++-Bibliothek, existieren auch in der Komponentenbibliothek MultiIndexsets, die aus einem oder mehreren einzelnen Indexsets bestehen. Die einzelnen Teil-Indexsets können über die Eigenschaft `IndexSet` ausgelesen oder geschrieben werden. Ebenso existieren Eigenschaften für das Auslesen von Anzahl und Größe der Indexsets und einzelner Elemente. Da das `IMOMultiIndexSet`-Interface von den Interfaces `IMOCustomEnum` und `IEnumVARIANT` abgeleitet ist, stehen auch deren Enumerator-Funktionalitäten zur Verfügung. Soll ein `MOMultiIndexSet` zur Iteration, etwa in einer `For…Each`-Schleife genutzt werden, so können Iterationsfilter gesetzt werden (`SetEnumFilter`), damit nur über die von der Maske des Filters zugelassenen Bereiche iteriert wird. Beispiel:

```
Dim ms as New MOMultiIndexSet
Dim s as New MOIndexSet
ms.InitIndexSets(s.Set(2005,2006,2007),s.Set("Jan","Feb","Mrz"))
ms.SetEnumFilter(s.Set(),"Jan")
For Each i in ms
…
Next
```

Das zweidimensionale MultiIndexSet `ms` besteht aus neun Elementen (*2005.Jan*, *2005.Feb* usw.). Der angegebene Filter für das erste IndexSet ist eine leere Menge, was bedeutet, dass keine Filtermaske angewandt wird und daher alle Elemente des Sets in den Enumerator einbezogen werden. Für das zweite Indexset besteht die Filtermaske nur aus einem einzigen Element, wodurch nur das Element „Jan“ zur Iteration benutzt wird. Somit läuft die Schleife über die drei Elemente *2005.Jan*, *2006.Jan* und *2007.Jan*.

Wie alle anderen mehrfach indizierbaren Komponenten, spannen sich MultiIndexSets über das kartesische Produkt der beteiligten Indexmengen auf. Zwar wurden bei der Implementati-

on intern bereits Vorkehrungen getroffen, auch dünn besetzte Indexsets (d.h. Teilmengen des kartesischen Produkts) abbilden zu können, aus Gründen der Vereinfachung wurde dieses Feature allerdings im vorliegenden Entwurf zunächst ausgespart.
Das Interface `IMOMultiIndexSet` vererbt darüber hinaus seine Eigenschaften und Methoden an die Interfaces der mehrfach indizierbaren Komponenten `MODataArray`, `MOVarSet` und `MOResSet`, so dass das Index-Handling aller mehrdimensionalen Objekte homogen ist.

MODataArray

Diese Komponente speichert Daten vom Typ Integer, Double, String oder Variant als ein- oder mehrdimensionales Array, das auf Basis eines MultiIndexsets beliebig alphanumerisch indizierbar ist. Die Komponente besitzt darüber hinaus Grundfunktionalitäten der Matrizenarithmetik, wie Multiplikation mit einem Skalar, Matrixmultiplikation, und Transposition. Ferner kann der Inhalt eines anderen `MODataArrays` oder eines Safearrays inklusive der Dimensionierung geklont werden.
Auf die einzelnen Werte des Arrays kann lesend und schreibend auf mehrere Arten zugegriffen werden: Einerseits unter Angabe der einzelnen Indexwerte (bis maximal fünf Werte mit Eigenschaft `Value`), oder unter Angabe eines zusammengesetzten Indexnamens der Art *"Indexwert1.Indexwert2...IndexwertN"* (Eigenschaft `ValueByFullIndexName`), durch Angabe eines die passenden Werte enthaltenden `MOTuple` oder durch Angabe der numerischen Indexsetwerte (`ValueByNumIndex`).
Zur Iteration über ein `MODataArray` stehen die Funktionalitäten der geerbten Enumerator-Interfaces zur Verfügung.

MOVarSet und MOResSet

In nahezu allen größeren Modellen kommen ein- oder mehrfach indizierte Variablen und Restriktionen vor, die über `MOVarSet` und `MOResSet` im Programmcode abgebildet werden können. Die Variablen- und Restriktionsattribute können für einzelne Variablen oder Restriktionen gelesen und geschrieben werden. Zum Beispiel:

```
myVarSet.UB(2005, "Jan") = 123
```

Es kann aber auch auf ganze Gruppen mit den schon zuvor erwähnten Filtermechanismen zugegriffen werden. So setzt etwa

```
myVarSet.UB( , "Jan") = 123
```

in einem zweidimensionalen, über Jahre und Monate indizierten Variablenset die Januar-Obergrenzen aller Jahre auf 123. Nach dem gleichen Prinzip können auch Werte ausgelesen werden, wie z.B.

```
myMODataArray = myVarSet.UB( , ”Jan”)
```

Hier werden die Werte der Obergrenzen der durch den Filter bezeichneten Variablen in ein `MODataArray` übertragen, das automatisch entsprechend dimensioniert und mit den passenden Indizes versehen wird. SafeArrays müssen bereits vorher ausreichend dimensioniert sein.
Ebenso wie für ein einzelnes Restriktionsobjekt vom Typ `MORes` kann der Restriktionstyp (<=, =, =>) für alle Restriktionen eines `MOResSet` mit einer einzigen Anweisung (`SetGE` etc.) gesetzt werden, und es können mit `Add` und `Del` für alle Restriktionen Variablen und Koeffizienten hinzugefügt, bzw. gelöscht werden.
Um ein einzelnes `MOVar`- oder `MORes`-Objekt für ein bestimmtes Element eines `MOVarSet` oder `MOResSet` zu erhalten, gibt es die Eigenschaften `Var` und `Res`, die als Parameter kommagetrennte Indexwerte, ein `MOTuple` oder einen vollständigen Indexnamen akzeptieren.

Enumerator-Komponenten

Für alle Komponenten mit aufzählbaren Elementen gibt es zugehörige Enumerator-Komponenten, die vor allem der Unterstützung von `For…Each`-Schleifen in Visual Basic dienen (`IEnumVARIANT`) und fernerhin über das Interface `IMOCustomEnum` das Durchlaufen von Schleifen mit vor- oder nachgelagerter Bedingungsprüfung erleichtern (in Visual Basic: `Do While` und `Loop Until`).
Die Benutzung des `IEnumVARIANT`-Interfaces und die Instanzierung der Enumerator-Komponente, die die Methoden dieses Interfaces implementiert, erfolgt in VB für den Benutzer völlig transparent, so dass beispielsweise folgende Syntax möglich ist:

```
Dim v as MOVar
For Each v In myMOVarSet
    Debug.Print v.Name
Next
```

Hierbei werden alle Variablen eines zuvor angelegten Variablensets durchlaufen und deren Name im Debug-Fenster ausgegeben. Intern geschieht dabei Folgendes: Zunächst wird durch

Aufruf der versteckten MOVarSet-Methode get__NewEnum eine neue Instanz einer MOVarSetEnum-Komponente erzeugt. Diese neue Instanz besitzt ein IEnumVARIANT-Interface, mit dessen Methoden (Reset, Next usw.) dann über das Variablenset iteriert wird. Dieser intern kompliziertere Mechanismus wird von der Visual Basic Runtime automatisiert und vor dem Benutzer verborgen.
Außer von IEnumVARIANT, sind die Interfaces aller MultiIndexset- und Enumerator-Komponenten weiterhin von IMOCustomEnum abgeleitet und können daher mit dessen Methoden und Eigenschaften iteriert werden, was vor allem für Do…Loop-Schleifen in Visual Basic vorteilhaft ist:

```
myMOVarSet.MoveFirst
Do While Not myMOVarSet.IsBeyondLast
     Debug.Print myMOVarSet.CurrentVar.Name
     myMOVarSet.MoveNext
Loop
```

Der in MOVarSet integrierte Custom-Enumerator wird dabei zunächst auf die erste Position des Sets gesetzt und dann so lange durchlaufen, bis er hinter dem letzten Element angelangt ist. Die Eigenschaft CurrentVar liefert das jeweils aktuelle MOVar-Objekt. Mit CurrentTuple, einer aus dem Eltern-Interface IMOMultiIndexSet geerbten Eigenschaft, könnte während der Iteration auch auf das jeweilige Index-Tupel zugegriffen werden.
Neben der direkten Iteration über ein MODataArray, MOVarSet oder MOResSet können für diese Komponenten auch die zugehörigen Enumeratoren benutzt werden. Diese enthalten lediglich eine Kopie der Indizes und ggf. eine Filtermaske und werden vor allem bei verschachtelten Iterationen erforderlich. Etwa:

```
Dim e1 as New MOVarSetEnum, e2 as NewMOVarSetEnum
Dim v1 as MOVar, v2 as MOVar
e1.Init(myMOVarSet) : e2.Init(myMOVarSet)
For Each v1 in e1
  For Each v2 in e2
    If v1.IPSol = v2.IPSol …
  Next
Next
```

Hier wird mit zwei Enumeratoren über das selbe, zuvor bereits indizierte Variablenset iteriert, und in der verschachtelten Schleife werden die IP-Lösungswerte der Schleifenvariablenobjekte miteinander verglichen. Besonders knappe und intuitive Iterationsmöglichkeiten bieten sich bei der zusätzlichen Benutzung von Filtermasken, die bei Initialisierung der Enumerator-Komponenten gesetzt werden können. So initialisiert beispielsweise folgende Anweisung einen Enumerator, der nur über die ersten Quartalsmonate eines mit Jahren und Monaten zweidimensional indizierten Restriktionssets läuft:

```
Dim e as New MOResSetEnum
Dim s as New MOSet    ' Hilfsset
e.Init(myMOResSet, ,s.Set("Jan","Feb","Mrz"))
```

Das Vererbungsschema (s. Anhang 5) sieht vor, dass die Interfaces `IMODataArrayEnum`, `IMOVarSetEnum` und `IMOResSetEnum` die Eigenschaften und Methoden von `IMOMultiIndexSetEnum` erben, was z.B. die Methoden zum Setzen und Auslesen der Filter betrifft. `IMOMultiIndexSetEnum` hat Funktionalitäten von `IMOCustomEnum` geerbt, das seinerseits von der universellen OLE-Enumeratorschnittstelle `IEnumVARIANT` beerbt wurde.
Abschnitt 8.1 zeigt noch mehr Beispiele, die die vielfältigen Einsatzmöglichkeiten der hier beschriebenen Komponenten weiter verdeutlichen.

6.4.4 .NET-Komponentenbibliothek

Eine .NET-Modellierungskomponentenbibliothek zielt primär auf die Programmiersprachen VB.NET und C# ab, andere .NET-Sprachen wie Delphi.NET könnten aber ebenfalls durch entsprechende Module abgedeckt werden. Für managed C++ ist die unmanaged C++-Klassenbibliothek aus Abschnitt 6.4.2 dank der fast schrankenlosen Interoperabilität dieser C++-Versionen direkt nutzbar.
Was die COM/.NET-Interoperabilität anbelangt, muss Folgendes bedacht werden: Man unterscheidet zwischen *chunky* und *chatty* Interfaces. ([IDN04] verweist auf mehrere Autoren/Fundstellen, die die Begriffe diskutieren): Ein *chunky* Interface stellt größere Funktionalitätseinheiten mit wenigen Methodenaufrufen zur Verfügung, während ein *chatty* Interface zur Bereitstellung einer vergleichbaren Funktionalität sehr viel mehr Interaktion mit dem Interface-Client benötigt. Beide Interfacetypen können situativ ein sinnvolles Designmuster sein. Liegen allerdings Client und Interface in unterschiedlichen logischen Domänen (etwa bei Webservices oder bei COM/.NET-Interoperabilität), so muss bedacht werden, dass jede Über-

schreitung der Domänengrenze Overhead bedeutet, der bisweilen erheblich sein kann. Eine Nutzung der COM-Modellierungsbibliothek ist zwar dank der Interoperabilitätsmechanismen aus .NET möglich, würde aber sehr hohe Performanceeinbußen mit sich bringen, da die Modellgenerierung mit den COM-Komponenten äußerst „chatty“ ist. Die .NET-Anwendung MOPS Studio benutzt dennoch die COM-Komponenten zur internen Modellverwaltung, da bisher noch keine .NET-Modellierungsbibliothek erstellt wurde. Allerdings erwachsen in MOPS Studio daraus keine wesentlichen Performancenachteile, da MOPS Studio die COM-Interfaces sehr „chunky“ benutzt und keine Modellgenerierung im Code betreibt. So ist beispielsweise der Interoperabiblitätsoverhead bei einem einzelnen Aufruf der Methode `ReadAMPLModel` sehr gering.

Für eine effiziente Modellgenerierung unter .NET ist es allerdings erforderlich, eine eigene .NET-Modellierungsbibliothek bereitzustellen, die keinen oder möglichst wenig Interoperabilitätsoverhead produziert. Zumindest nach außen sollten daher echte .NET-Komponenten bereitgestellt werden, intern könnten Teile der gemeinsamen Modellierungsklassen wieder verwendet werden. Zur Implementation eines solchen Designs könnte managed C++ verwendet werden, da dort eine Vermischung von managed und unmanaged Code am einfachsten und flexibelsten möglich ist. Eine solche .NET-Bibliothek entspricht weitgehend der C++-Bibliothek des OO-Interface Layers (Abschnitt 6.4.2), weshalb hier Klassen, Methoden und Eigenschaften nicht noch einmal aufgeführt werden sollen. Einige Besonderheiten des Schnittstellendesigns bei .NET-Sprachen haben aber Auswirkungen auf Designdetails der .NET-Modellierungsbibliothek und sollen daher kurz erwähnt werden:

- **Operatorüberladung**: Ebenso wie C++ können in C# Operatoren überladen werden, was ein Kernfeature für die Annäherung des Programmcodes an die algebraische Notation ist (s. Abschnitt 6.4.2.1). VB.NET unterstützt Operatorenüberladung erst ab Version 2005. Demzufolge muss eine .NET-Modellierungskomponentenbibliothek entsprechende Klassen für den Aufbau von Restriktionen und Zielfunktion mit überladenen Operatoren bereithalten (Terme, lineare Ausdrücke etc.).
- **Optionale Parameter**: VB.NET unterstützt optionale Parameter, C# jedoch nicht, bzw. nur mit einem Workaround (über `System.Type.Missing`). Dies muss bei der Umsetzung von Methodenaufrufen mit optionalen Parametern berücksichtigt werden (z.B. bei Klasse `CModel`).
- **Parametrisierte Eigenschaften**: Werden in VB.NET unterstützt, in C# jedoch nicht. In C# muss auf Eigenschaften, die Parameter akzeptieren über Accessormethoden zugegriffen werden. Die COM-Bibliothek zeigt, wie man parametrisierte Eigenschaften

sehr gut zur Erreichung einer intuitiven Syntax einsetzten kann (vgl. Abschnitt 6.4.3). Konstrukte wie z.B. `myModel.NZ(1,2)=4711` zum Setzen eines Nonzeros wären daher nur in einer Komponentenbibliothek für VB.NET möglich.

Zur optimalen Ausnutzung syntaktischer Besonderheiten einzelner .NET-Sprachen würde sich daher folgendes Design empfehlen: Ein für alle .NET-Sprachen gemeinsamer Kern stellt das Bindeglied der .NET-Sphäre mit den darunter liegenden nativen Ebenen des Modelmanagement Layers und des numerischen Kernels dar. Zusätzlich existieren sprachspezifische Module als sehr dünner Kapselung, die Ausprägungen für die jeweiligen Sprachen darstellen.

6.5 OSI

Das *Open Solver Interface (OSI)* ist ein Teil des Projekts COIN ([COIN06]), in dem Solver, Tools und Schnittstellen als Open Source entwickelt werden. Der Zweck von OSI ist die Bereitstellung einer einheitlichen Schnittstelle für unterschiedliche mathematische Solver zur Förderung der Austauschbarkeit in Applikationsumgebungen. OSI ist Plattform übergreifend in ANSI/ISO C++ realisiert, und es existieren Anbindungen für über zehn Solver (z.B. CPLEX, XPRESS, CLP). Wie in Abbildung 27 dargestellt, arbeitet der User Code dabei primär mit einer generalisierten Klasse `OsiSolverInterface`, von der die solverspezifischen Klassen als Wrapper für die jeweiligen Solver abgeleitet sind. `OsiSolverInterface` bietet eine Vielzahl von Methoden, wie etwa zum Einlesen einer MPS-Datei, Starten des Optimierungslaufs oder Auslesen der Lösung.

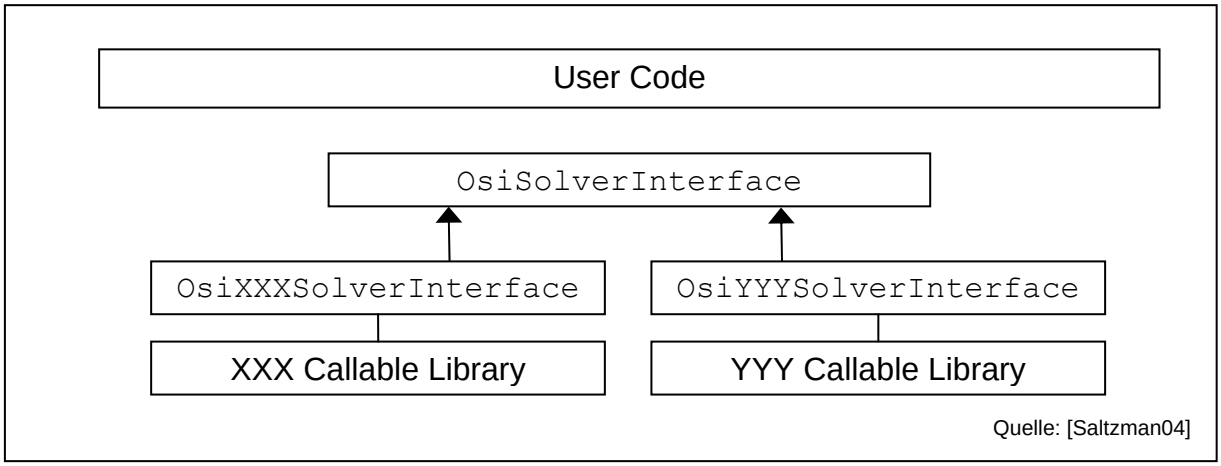

Abbildung 27: Open Solver Interface

Das Modell wird als gleichförmige Modellrepräsentation `OsiModel` vorgehalten, von der spezifische Modellausprägungen für unterschiedliche Modelltypen abgeleitet sind (z.B. für

lineare oder quadratische Modelle). Außerdem soll das algorithmische Verhalten der Solver vereinheitlicht werden, indem die solverspezifischen Implementationen unterschiedlicher Lösungsalgorithmen in uniforme Klassen gekapselt werden (etwa `OsiSimplexAlgorithm`, `OsiBarrierAlgorithm`) und gemeinsame Parameter definiert werden.

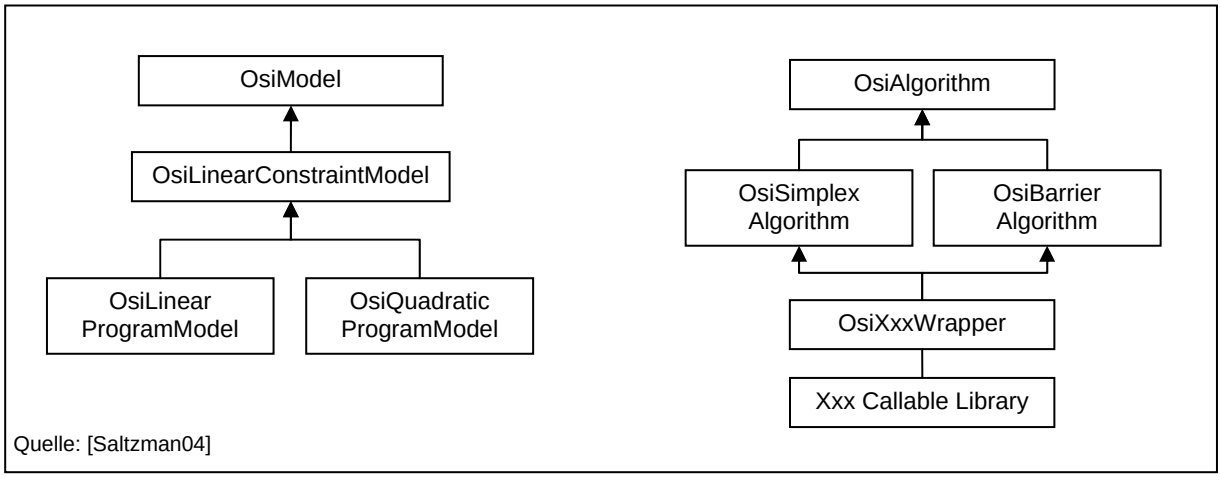

Abbildung 28: OSI Beispiele Klassenhierarchie

Trotz bestimmter Ähnlichkeiten unterscheidet sich OSI von dem in dieser Arbeit entworfenen Schnittstellenkonzept in mehrfacher Hinsicht: OSI zielt ausschließlich auf die einheitliche Anbindung unterschiedlicher Solver ab. Das hier konzipierte System sieht zwar auch die Anbindung verschiedenartiger Solver vor, sein primäres Ziel ist aber darüber hinaus die Vereinfachung der Modellgenerierung aus unterschiedlichen Framework- und Programmierkontexten und die Steigerung der Effizienz der Modellgeneratorerstellung. OSI ist auf C++ als Modellgeneratorensprache beschränkt, während das hier beschriebene System Schnittstellen für verschiedene Sprachen vorsieht. Auch beachtet OSI lediglich die Programmierschnittstellen, während sich das hier entworfene Konzept bis zu den Benutzerschnittstellen hinzieht und mit MOPS Studio sogar sehr umfangreiche Implementationsleistungen auf dieser benutzerbezogenen Ebene leistet. Trotz des gemeinsamen Ziels der Anbindung unterschiedlicher Solver stehen beide Systeme somit nicht in unmittelbarer Konkurrenz zueinander.

6.6 Schnittstellen zu Modellierungssprachen

Schnittstellen zu Modellgenerierungssprachen können in zwei unterschiedlichen Ausgestaltungskonzepten realisiert werden:

- “Modeling language consumes solver“: Bei dieser Variante ist die Modellierungssprache in Form einer Modellierungssprachenbibliothek oder einer Integrierten Modellie-

rungssprachenumgebung die steuernde Einheit, die den Solver über dessen Standardinterfaces anspricht. Logisch betrachtet, nimmt die Modellierungssprache die Rolle eines Clients- und der Solver die des Servers an.

- "Solver library consumes modeling language": Hierbei spricht die Solverbibliothek in Client-Rolle die Modellierungssprache als Server an. Die Modellierungssprache muss dazu ein öffentliches Interface besitzen.

Das erste Konzept ist typischerweise in Systemen umgesetzt, bei denen die Modellierungssprache im Vordergrund steht und verschiedene Solver in austauschbarer Weise als Module oder Plug-Ins benutzt werden können. Beispiele hierfür sind MOSEL mit der XPRESS IVE oder MPL mit MPL Studio.
Das hier umrissene Middlewarekonzept sieht hingegen einen hybriden Ansatz vor, bei dem die Middleware primär eine steuernde Client-Rolle annimmt, aber auch in die Server-Rolle wechseln kann, falls die Modellierungssprache dies unterstützt und bestimmte Modelle es erfordern.

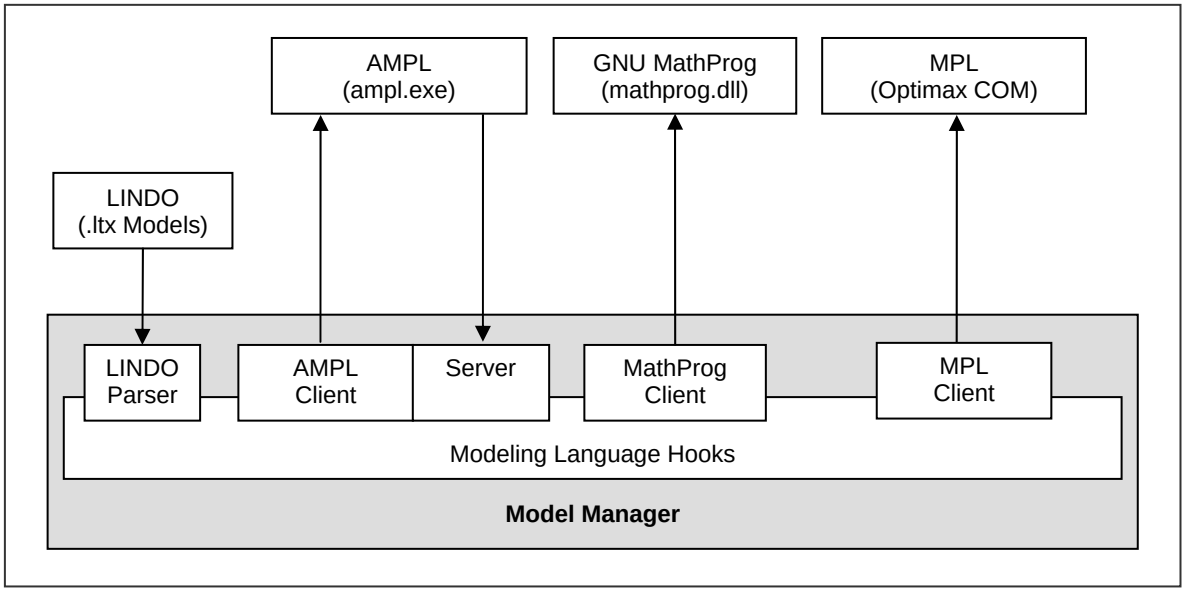

Abbildung 29: Modellierungssprachenanbindung

Wie in Abbildung 29 skizziert, ist eine erweiterbare Modellierungssprachenanbindung vorgesehen, die aus einer Reihe von Modulen (*Hooks*) besteht, die die Schnittstellen der einzelnen Sprachen ansteuern und Modelle in das interne Format des Model-Managers übertragen. Bisher wurden Hooks für die folgenden vier Modellierungssprachen implementiert:

- AMPL

  AMPL liegt in Form einer ausführbaren Datei vor und besitzt lediglich eine Komman-

dozeilenschnittstelle. Der AMPL-Hook kann diese Schnittstelle bedienen, indem er mittels Interprozesskommunikation der Schnittstelle Anweisungsfolgen gibt, die AMPL veranlassen, ein Modell (.mod) und eine beliebige Anzahl von zugehörigen Daten-Files (.dat) zu lesen und daraus ein so genanntes Stub-File zu generieren. Mithilfe einer ebenfalls von AMPL zur Verfügung gestellten Bibliothek können solche Stub-Files anschließend ausgelesen und vom Model-Manager in sein internes Format transformiert werden. In diesem Fall ist die AMPL.EXE ein Server, der vom Model-Manager als Client gesteuert wird. AMPL besitzt allerdings – im Gegensatz zu den anderen drei Modellierungssprachen – prozedurale Sprachfeatures, die den Aufruf eines Solvers aus der Modellierungssprache selbst erlauben. So können iterative oder sogar rekursive Abfolgen von Optimierungsläufen formuliert werden. In diesem Fall vertauschen sich die Rollen dergestalt, dass nun AMPL der Client und der Model-Manager der Server ist. Für diese Konstellation existiert das Middleware Modul *moampl.exe*, das von AMPL als Solver/Server angesprochen wird, wenn AMPL den *solve*-Befehl im Modellcode ausführt.

- MathProg

  MathProg implementiert als Open Source-Lösung ein Subset des AMPL Sprachumfangs, unterscheidet sich aber bezüglich seiner Solverschnittstellen stark von AMPL. Anders als AMPL liegt MathProg als statische oder dynamische Bibliothek vor, weshalb es zur Ansteuerung keiner Interprozesskommunikation bedarf. Weiterhin erfolgt die Modellübergabe nicht über Stub-Files, sondern über eine prozedurale, aus rund 100 Funktionen bestehende Schnittstelle, die für die Zusammenarbeit mit der Optimierungsmiddleware von uns noch etwas vereinfacht wurde. Eine hybride Schnittstellenarchitektur (Model-Manager als Client und Server), wie bei AMPL, ist bei MathProg weder möglich noch erforderlich, da zwar ein Befehl *solve* in der Sprache existiert, dieser aber nur zum Aufruf des GLPK-Solvers benutzt werden kann.

- LINDO

  Im Vergleich zu AMPL oder MPL ist LINDO eine sehr einfache Modellierungssprache, die eher ein Dateiformat, als eine Sprache darstellt, da wichtige und typische Merkmale von Modellierungssprachen (Indizierung, Datenhandling etc.) fehlen. Aufgrund dieser Einfachheit wurde der LINDO-Hook in der Middleware-Bibliothek als autonomer Parser konzipiert, der zum Lesen von .ltx-Dateien keine Fremdsoftware benötigt.

- MPL

  Als einzige der verbreiteten Modellierungssprachen liegt MPL in Form einer COM-Bibliothek („Optimax“) vor. Ein MPL-Interfacemodul muss daher zur Kommunikation clientseitig COM benutzen. Die Optimax-Bibliothek bietet Methoden und Eigenschaften zum Einlesen einer MPL-Datei (.mpl) und der dort referenzierten Datendateien (.dat), sowie umfangreiche Möglichkeiten, um auf die daraus generierte Matrix zuzugreifen. Der Datenaustausch geschieht dabei komplett innerhalb des Hauptspeichers und nicht über Zwischendateien wie bei AMPL. Umgekehrt kann aber auch MPL in der Rolle des Clients einen Solver als Server aufrufen. Dieser Weg ist für die hier beschriebene Middleware zwar nicht relevant, die MPL-eigenen Solverschnittstellen benutzen aber angebundene Solver wie CPLEX oder MOPS auf diese Weise. Für MOPS existiert daher nicht nur die MPL-Anbindung über die hier beschriebene COM-Bibliothek, sondern zusätzlich wurde auch eine MPL-spezifische, direkte MOPS-Schnittstelle entwickelt. Hierzu mussten in MOPS einige für MPL obligatorische Voraussetzungen geschaffen werden (z.B. Node-, Iteration- und Log-Callbacks, bestimmte Routinen zum Setzen und Lesen von Parametern etc.).

6.7 MOPS Studio als endbenutzerbezogene Schnittstelle

Das hier entworfene Middlewarekonzepts erstreckt sich nicht nur auf die Programmierschnittstellen, sondern zieht auch die Endbenutzerschnittstellen mit ein. In diesem Zusammenhang wurde als interaktive Endbenutzerschnittstelle das integrierte Modellierungssystem *MOPS Studio*[1] entwickelt, das neben der Komponentenbibliothek den zweiten Schwerpunkt der erfolgten Programmierarbeiten darstellt. Ziel war dabei die Umsetzung der in 5.3.3 aufgeführten Anforderungen an Endbenutzerschnittstellen. Innerhalb der durch begrenzte Ressourcen gesetzten Grenzen stellt MOPS Studio ein stabiles und komfortables integriertes Modellierungssystem dar, das sich bereits in der Lehre mehrfach bewährt hat und nunmehr an der Schwelle zu einem kommerziellen Produkt steht.

MOPS Studio ermöglicht mit seiner modernen grafischen Benutzeroberfläche die Erstellung von Modellen in den Modellierungssprachen AMPL, MathProg, LINDO und MPL und deren Optimierung (derzeit nur mit MOPS, weitere Solver folgen später). Somit unterstützt das System von den in [Geoffrion89] beschriebenen drei Hauptformen der Integration (*model integration*, *solver integration* und *utility integration*) nur die beiden Letztgenannten. Modellinteg-

[1] Der Name „MOPS Studio“ kommt vom primär eingesetzten Solver MOPS. Da das System aber auf der Modellierungsmiddleware aufsetzt, sind auch andere Optimierer anbindbar.

ration ist weder vorhanden noch geplant, da das System ausschließlich für LP-/MIP-Modelle ausgelegt ist.
Autoren wie [Tsai01] sehen Parallelen zwischen dem Modellmanagement in integrierten Modellierungssystemen und Datenbankmanagement, und sie schlagen daher Features für Modellierungssysteme vor, die das Datenmanagement in den Vordergrund rücken und an Datenbanksystemfeatures angelehnt sind, wie z.B.:

- Modelbrowser für interaktive Modellabfrage
- Erzeugung und Nutzung einer (semi-)automatisch erstellten Modelldokumentation
- Modellabfragemöglichkeiten mittels einer „Model-Query-Language“, die ähnlich einer Database-Query-Language funktionieren soll
- Modell- und Modelldaten-Manipulation mittels Befehlsanweisungen, die ähnlich dem SQL-Befehl „update“ Modifikationen durchführen können
- Reporting-Funktionen, die auf Modeloutput basieren, der einer Modeloutput-Datenbank entnommen ist (vgl. auch [Muhanna94])
- Trennung von Modell („Schema“) und Daten („Instanz“)
- „Data-Dictionary“ für Modelldaten, ähnlich Metadaten in einem Datenbanksystem

Von Vorschlägen wie diesen konnten nur wenige in das Design von MOPS Studio einfließen, wie etwa die Trennung von Modell und Daten. Für die Übrigen ist die Mitwirkung der Modellierungssprache unverzichtbar, was allerdings die derzeit eingebundenen Sprachen nicht leisten. Bei strukturierter Modellierung (vgl. [Geoffrion87], [Geoffrion89b]) mit der Sprache *SML* (vgl. [Geoffrion92]) hingegen würden einige dieser Features möglich, wie etwa die Dokumentationserzeugung aus dem Modelltext.
Das Modellierungssystem MOPS Studio zeichnet sich durch die folgenden Features aus:

- Multithreading-Optimierung. Parallele Läufe mehrerer Modelle sind möglich, obwohl MOPS als Solver dies „von Haus aus“ nicht unterstützt.
- Volle Ausführbarkeit von AMPL-Modellen, wobei MOPS Studio für AMPL auch die Rolle eines Optimierungsservers übernehmen kann (*Execute AMPL*).
- Intelligentes Syntax-Coloring für AMPL, LINO und MPL zur besseren Lesbarkeit des Modellcodes
- Zusammenfassung von Modell, Parametereinstellungen, Lösungsreports und weiterer Dateien, wie z.B. Notizen oder Debug-Files zu Projekten
- Lesen und Schreiben von MPS- und Triplet-Dateien
- Konvertierung beliebiger Modelle in MPS-, Triplet- oder LINDO-Format

- Ausgabe von gut lesbaren Debug-Modellen, zur Überprüfung der aus den Modellierungssprachen erzeugten Matrix
- Debugging-Unterstützung durch Suche und Anzeige von syntaktischen Fehlern in Modellen und teilweise Angabe von Korrekturmöglichkeiten
- Benutzer kann vor Optimierung syntaktischen Check laufen lassen.
- Umfangreiche Drag&Drop-Unterstützung für Modelle und sonstige Dateien
- Komplexe Benutzeroberfläche mit vielfachen Docking- und Splitting-Funktionalitäten ähnlich wie in Visual Studio 2005
- Visualisierung von Zwischenergebnissen während der Optimierung
- Benutzerinduzierte Abbruchmöglichkeiten der Optimierung (zuvor in MOPS nicht möglich)
- Automatische Updates über das Internet
- Grafische Oberfläche zur Einstellung der MOPS-Parameter mit Erläuterungen
- Komfortabler und schneller, auch für größere Modelle geeigneter Editor
- Komplett überarbeitete MOPS-Hilfe (CHM), teilweise mit Verknüpfungen aus MOPS Studio zu spezifischen Hilfe-Themen
- Lauffähigkeit unter Windows 2000, XP und Vista
- 32- und 64-Bit Versionen (XP)
- Kostenlose und unlimitierte Version für Lehrzwecke
- Distribution als InstallShield-Executable inkl. Beispielmodelle, Hilfe etc.

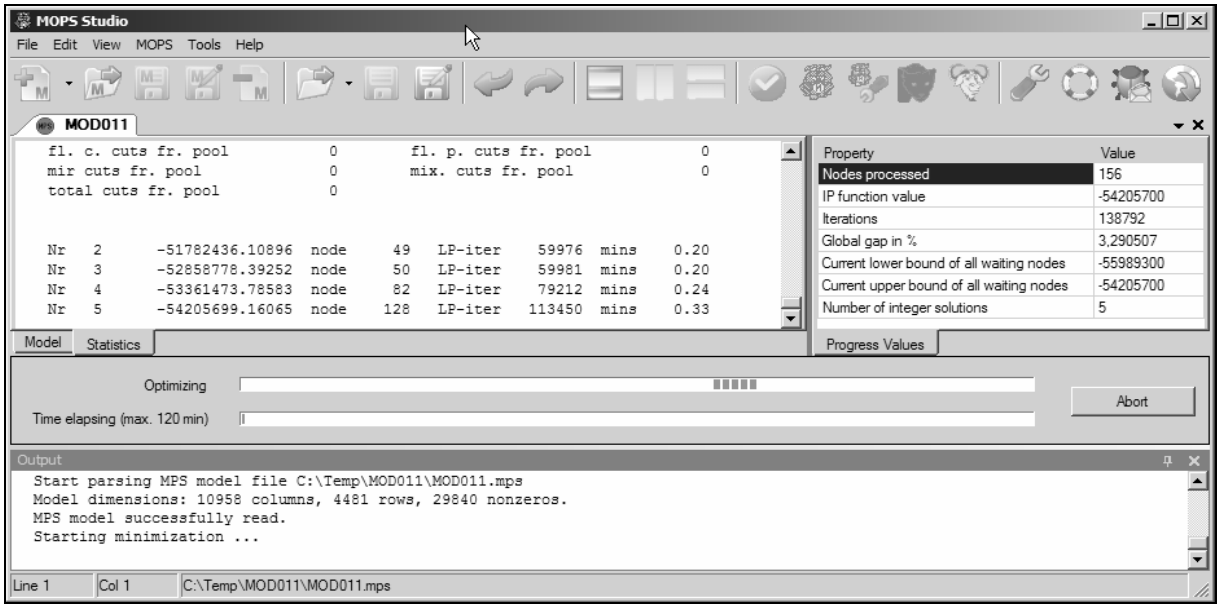

Abbildung 30: MOPS Studio (1)

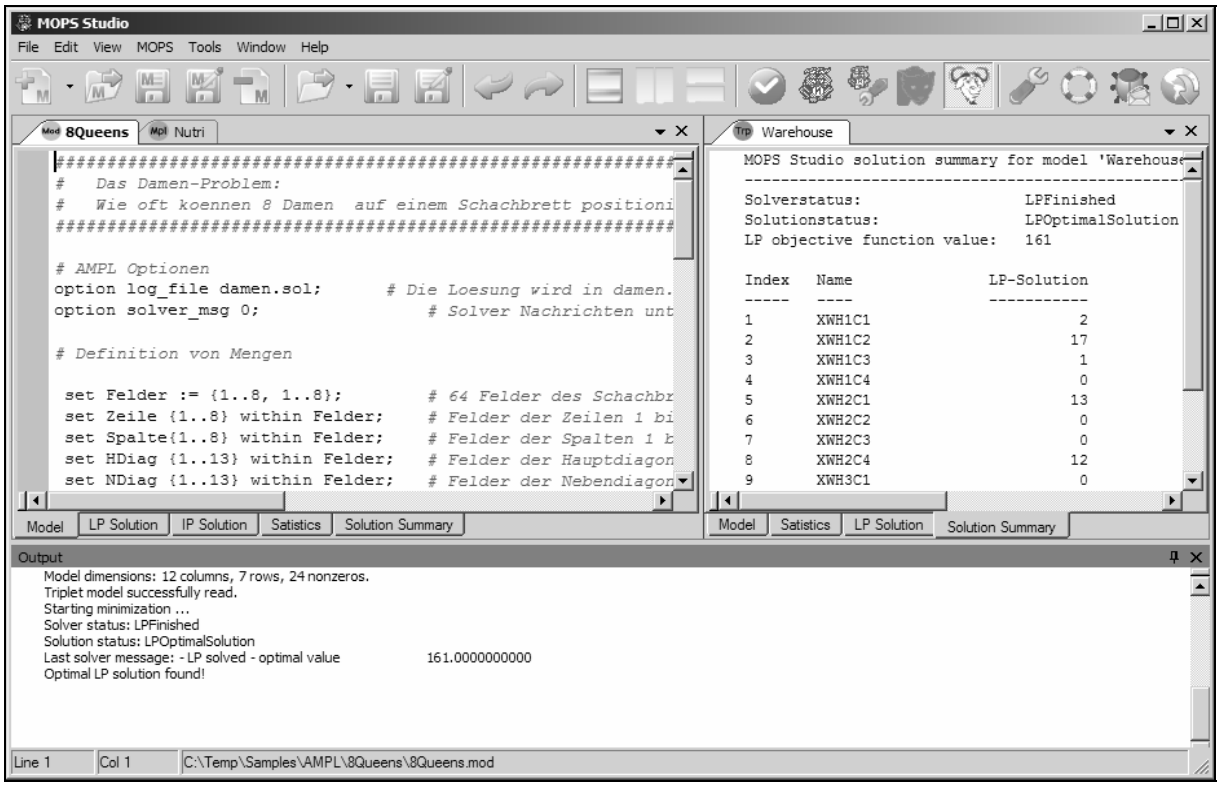

Abbildung 31: MOPS Studio (2)

Abbildung 30 und Abbildung 31 zeigen, die Oberfläche von MOPS Studio mit geöffneten Modellen und Status-Output während und nach erfolgter Optimierung.

In 7.4 werden einige Details der Implementation von MOPS Studio erläutert, und Kapitel 8 bewertet in wieweit MOPS Studio den allgemeinen Anforderungen an integrierte Modellierungssysteme aus 5.3.3 entspricht.

6.8 Optimierungsschnittstellen im Rahmen Service Orientierter Architekturen

Das Konzept der Service Orientierten Architektur (SOA) ist eines der wichtigsten Paradigmen der aktuellen Informatik[1]. SOA ist eine Architektur der Verknüpfung von Geschäftsprozessen mit IT-Ressourcen, wobei jene in abstrakter Form als Dienste aufgefasst werden, die in loser Kopplung aus unterschiedlichen Geschäftsprozessen in Anspruch genommen werden können. Ein formaleres SOA-Referenzmodell findet man u.a. auf [OASIS06]. Dabei ist SOA nicht an eine spezielle Technologie gebunden, und zur Implementation können unterschiedliche Protokolle und Interfacetypen eingesetzt werden (z.B. DCOM, RPC, SOAP, WebServices,

[1] Die Literatur zu SOA ist sehr umfangreich. Als Einführung und Übersicht seien [Erl05] und [Melzer07] herausgegriffen.

CORBA). Häufig wird das prinzipiell abstrakte und technologieunabhängige Konzept SOA allerdings mit den Standards XML, WebServices und SOAP in Verbindung gebracht.

„Traditionelle“ komponentenbasierte Designansätze beschäftigen sich häufig nicht mit Service Orientierten Architekturen, weil jene erst später, insbesondere mit dem Aufkommen von Webservices, populär wurden. Dennoch sind die grundlegenden Ideen von (objektorientierten) Komponentenarchitekturen und SOA sehr ähnlich (vgl. [Apperly03]): In beiden Fällen ist das Anbieten eines Dienstes durch ein standardisiertes Interface mit verbindlichen Kontrakten unter Vermittlung eines Frameworks charakteristisch. Der Entwurf einer Service Orientierten Architektur bedeutet jedoch mehr, als lediglich bestehende Software als JavaBean oder .NET-Komponente zu verpacken und als WebService bereitzustellen ([Stojanovic05, S. 49]). So zeichnen sich SOA-Designs im Vergleich zu lokalen Komponentenarchitekturen insbesondere durch eine losere Kupplung, verbunden mit einer gröberen Granularität der Interfaces aus (vgl. [Kaye03]).

Überlegungen, auch Optimierungsressourcen webbasiert zugänglich zu machen, sind indes grundsätzlich nicht neu, so enthalten etwa bereits Beiträge von [Bhargava97], [Bhargava98] oder [Fourer01] Konzepte und Gesichtspunkte zum *Application Service Providing* von Optimierungsservices über das Internet. Ein reales Beispiel für die Zurverfügungstellung von Optimierungsressourcen über das Internet ist das Projekt NEOS ([Dolan02], [NEOS06]), das Modellupload und Optimierung auf den NEOS-Servern durch diverse Solver ermöglicht. Für NEOS existieren übrigens auch AMPL-Remoteschnittstellen, die neben weiteren webbasierten Optimierungsanwendungen in [Fourer01] beschrieben sind.

Das Aufkommen von WebServices und die zuweilen schlagwortartige Popularität des SOA-Konzepts hat jedoch die Betrachtung von Optimierungssoftware als konsumierbarer Dienst neu fokussiert, was sich auch an einer Reihe von Veröffentlichungen zu neuen Konzepten aus diesem Bereich niederschlägt (s. unten). Eine verteilte Optimierung, bei der Client (GUI, evtl. Modellgenerierung) und Server (Solver) im Rahmen einer SOA auf unterschiedlichen Maschinen liegen, kann folgende Vorteile haben:

- Höhere Performance durch Ausnutzung leistungsstarker Serverhandware
- Geringerer Installations- und Wartungsaufwand der Solver-Software
- Geringere Lizenzkosten als bei lokal verteilten Solver-Lizenzen

Eine Service Orientierte Architektur für verteilte Optimierung muss Standards festlegen für die Kommunikation zwischen Modellierungssprachen, Solvern, Problem-Analyzern, Simulations-Engines sowie Registry- und Katalog-Diensten (vgl. [Fourer06]). Abbildung 32 zeigt

das Grundprinzip einer WebService-Architektur für Optimierungssoftware: Ein Solver registriert seinen Service inkl. bestimmter Servicemerkmale (z.B. verarbeitbare Modelltypen wie etwa linear, quadratisch etc.) bei einem Registry-Dienst. Ein Client kann über diesen Registry-Dienst Adresse und weitere Informationen über den Optimierungsserver erfahren, anschließend die erzeugte Modellinstanz zur Optimierung an den Server übermitteln und von diesem die Lösung zurückbekommen.

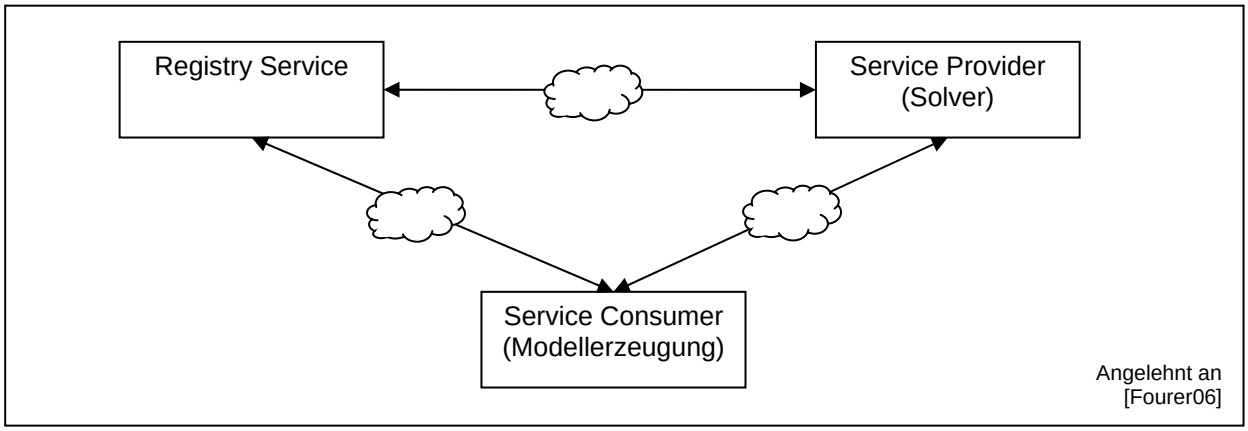

Abbildung 32: Optimierungssoftware als WebService

Für die Übertragung des Modells vom Client an den Service Provider (Solver) bieten sich XML-basierte Formate an, die in einem SOAP-Envelope gekapselt über HTTP übertragen werden können. Ein solcher Ansatz ist z.B. das von [Kristiansson01] vorgeschlagene Modellformat *OptML* und das von [Chang04] beschriebene Format *XML:LP* für LP- und MIP-Modelle. In die gleiche Richtung geht das in [Fourer03b] konzipierte *LPFML*, ein XML-Schema und eine zugehörige C++-Bibliothek zur Vereinfachung des Austauschs von LP-/MIP-Modellen zwischen Modellierungssystemen und Solvern. In [Fourer06b] werden die Grundgedanken von LPFM weiterentwickelt zu einen ganzen Bündel von XML- und Web-Service-basierten Protokollen zur verteilten Optimierung: Zunächst sei hier das offene Protokoll *OSiL* (Optimization Services instance Language) erwähnt, das im Gegensatz zu LPFM auch nicht-lineare Modelle darstellen kann. XML-Schemata werden im Rahmen von OSiL genutzt, um u.a. Verifikationen durchzuführen, z.B. ob ein Spaltenname ein String oder ein Index eine Integer-Zahl ist. Das Konzept Fourers umfasst neben OSiL zur Darstellung von Modellinstanzen weitere Protokolle wie etwa *OSrL* (Optimization Services result Language) zur Übertragung der Optimierungsergebnisse vom Solver zum Client oder *OSoL* (Optimization Services option Language) zur Übertragung der Solver-Optionen. Während die zuvor genannten Konzepte auf die XML-basierte Darstellung einer Modellinstanz abzielten, entwirft [Ezechukwu03a] ein Optimierungsframework, bei dem nicht einzelne Modellinstanzen, son-

dern das Modell in einer generischen Form getrennt von den zugehörigen Daten dargestellt wird. Im Mittelpunkt steht dabei die ebenfalls XML-basierte Sprache *AML* (Algebraic Markup Language, vgl. [Ezechukwu03b]). AML-Modelle werden mittels spezieller Transformatoren in das Zielformat des gewünschten Solvers gebracht, wobei die Modellinstanz erst nach dem Transformationsvorgang entsteht (Details siehe [Ezechukwu03a]). Als Gegenstück zu AML wird für die Rückübertragung der Lösungsergebnisse und anderer Solvermeldungen die Sprache *ORML* (Optimization Reporting Markup Language) vorgeschlagen, die in [Ezechukwu03c] beschrieben ist.

Die hier konzipierte Optimierungsmiddleware kann aus den zuvor genannten Ansätzen einige Prinzipien übernehmen, um eine Anbindung im Rahmen einer Service Orientierten Architektur zu ermöglichen. Dabei finden Modellgenerierung und Optimierung auf unterschiedlichen Rechnern statt, die in WebService-typischer Form über HTTP und SOAP kommunizieren. Das Modell wird lokal auf dem Client über die kontextspezifischen Interfaces (DLL, COM-Bibliothek, MOPS Studio etc.) im Model-Manager generiert und über das Netz an ein Modul übertragen, das den als WebService gekapselten Solver-Manager enthält. Dieser übergibt das Modell nach Transformation in das jeweils solverspezifische Format an den Optimierer. Eine Generierung des Modells über WebServices empfiehlt sich nicht, da das resultierende Interface zu „chatty“ wäre, d.h. eine zu feine Granularität hätte und daher erheblichen Kommunikationsoverhead verursachen würde. Zur Übertragung der Modellinstanz kann OSiL genutzt werden und für die Parameter- und Lösungsübertragung bieten sich OSoL bzw. OSrL an. Für die asynchrone Kommunikation zwischen Model-Manager und Solver-Manager sind Befehle ähnlich denen der *OShL* (Optimization Service hookup Language, vgl [Fourer06]) erforderlich, wie z.B. `knock(…)` zur Statusabfrage des Solvers oder `kill(…)`, um eine remote laufende Optimierung zu stoppen. Die OSxL-Protokolle dienen als Grundlage, können aber durchaus bedarfsgerecht abgewandelt werden, da sie hier nur zur Kommunikation innerhalb der Middleware dienen und keinen öffentlichen Charakter besitzen.

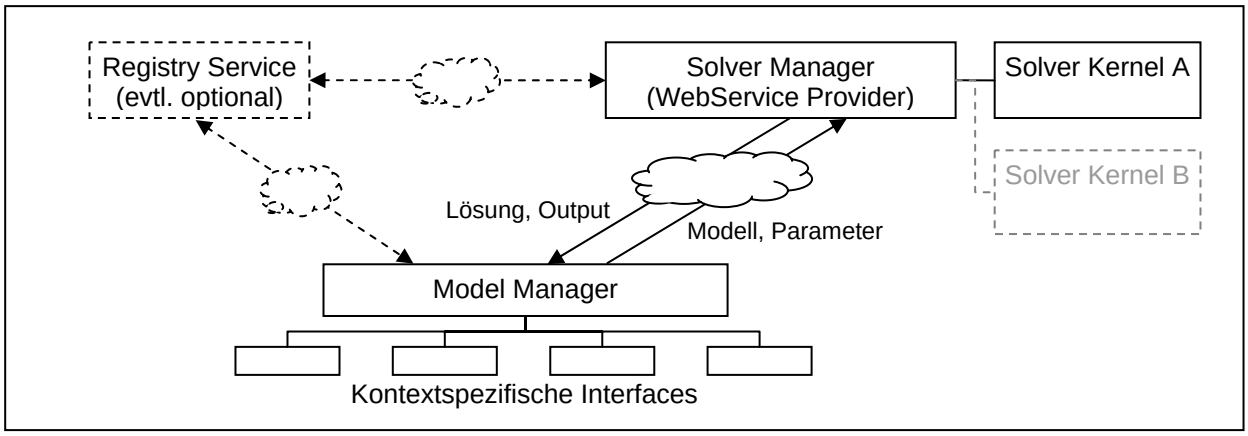

Abbildung 33: Optimierungsmiddleware und SOA

Der hier grob skizzierte Entwurf für eine WebService-basierte Erweiterung der Optimierungsmiddleware wurde noch nicht implementiert, da insbesondere Generierung und Parsing der OSxL-Formate, trotz Vorhandensein von XML-Hilfsbibliotheken, einen erheblichen Programmieraufwand darstellt. Testweise implementiert wurde hingegen eine DCOM-Version der Hauptkomponente (`MOModel`) der COM-Bibliothek, die ohne größere Veränderungen auch in MOPS Studio benutzt werden könnte, um verteilte Optimierung über DCOM zu realisieren.