7 Implementierung

Nachdem im vorigen Kapitel das Schnittstellenkonzept konzeptionell umrissen wurde, soll hier nun eine Auswahl an technischen und softwarearchitektonischen Details der vorgenommenen Implementationen erläutert werden.

7.1 Prozedurale Schnittstellen

Wie bereits zuvor erwähnt, wurde der für die prototypische Implementation benutzte Solver MOPS im Sinne eines Zwischenschritts im Bereich seiner prozeduralen Interfaces überarbeitet. Ziel war, neben der Schaffung von Voraussetzungen für die Einbindung in die Optimierungsmiddleware und das integrierte Modellierungssystem, ferner auch die Verbesserung der nativen Schnittstellen des Solvers im Sinne der Umsetzung des hier postulierten Grundsatzes der Einheitlichkeit der Schnittstellen, was zu einer Annäherung der IMR-Schnittstelle an das DLL-Interface führte.

7.1.1 DLL- und IMR-Interfaces

Wie in Kapitel 6.3 erwähnt, wurde ab MOPS Version 7.x die statische IMR-Schnittstelle dem bestehenden DLL-Interface angepasst, so dass die Funktionen beider Schnittsellen nunmehr einander weitgehend analog sind. Abbildung 34 zeigt diese neue Struktur.

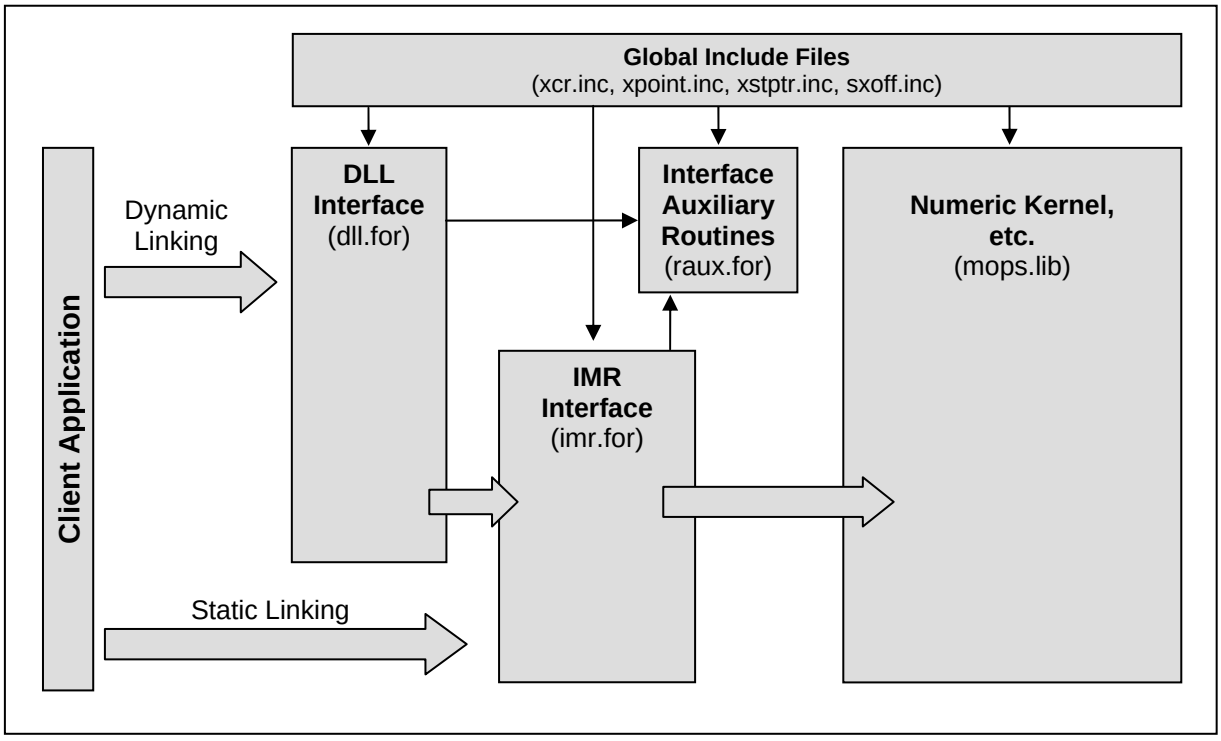


Abbildung 34: MOPS DLL- und IMR-Interfacearchitektur

Das DLL-Interface (dll.for) exportiert alle DLL-Funktionen und definiert

- Naming Convention (*Mixed Case*)
- Calling Convention (*stdcall*)
- Argument Passing Conventions (*ByValue* für skalare Input-Typen, *ByRefence* für alle anderen Typen)

Darüber hinaus erfolgen in dll.for im Wesentlichen nur Logging und Formatumwandlungen (z.B. C-Strings in Fortran-Strings etc.). Für die Erbringung der eigentlichen Funktionalität rufen fast alle DLL-Funktionen die analogen Routinen der IMR auf. Diese IMR-Routinen können bei statischem Linking aus einer Fortran-Clientapplikation auch direkt angesprochen werden, so dass die DLL-Schnittstellenebene übergangen wird. Sowohl dll.for als auch imr.for machen Gebrauch von einigen Interfacehilfsfunktionen (z.B. Namenssuche) die sich in raux.for befinden. Andere Interface-Routinen (wie z.B. MPS-File I/O etc.) sind mit den Routinen des Numerischen Kerns in der MOPS Kernel-Library *mops.lib* zusammengefasst. Eine klare Trennung von Interfaceschicht und Kernel ist daher bisher nur teilweise erreicht und könnte noch verstärkt werden.

7.1.2 Internes Speicherpointermanagement

MOPS benutzt für das Modell und fast alle Hilfsarrays intern einen großen zusammenhängenden Speicherbereich, der bei Aufruf der DLL-Funktion `AllocateMemory` angelegt wird, und aus dem eine Vielzahl von Arrays „herausgeschnitten“ werden. Dazu werden für jedes Array Pointer auf eine Startadresse innerhalb des großen Speicherblocks, der mit der Variablen `b` bezeichnet wird, berechnet und abgespeichert. Mithilfe eines solchen Pointers kann dann ein Array (etwa `xub` für die Upper Bounds) auf den entsprechenden Teilbereich in `b` gelegt werden, wie folgendes Fortran-Codebeispiel zeigt:

```
real*8, allocatable, save :: b(:)   ! Der große Speicherbereich
integer*4 xubz                      ! Pointer bei 32 Bit OS
real*8    xub(1)                    ! Upper Bounds Array
pointer (pxub, xub)
allocate (b(100000))                ! 800000 Bytes großen Bereich anlegen
xubz = loc(b(5000)))                ! Start bei relativer Position 5000
pxub = xubz                         ! Start von xub an Pos. 5000 in b
```

Das so auf `b` gelegte Array `xub` kann bei Deaktivierung der Compilereinstellung für Array Bound Checks auch jenseits der Position 1, bis zum Erreichen des oberen Endes von `b` benutzt werden, ohne dass Speicherverletzungen aufträten. Dieses Allokationsprinzip zieht sich durch den gesamten Sourcecode von MOPS und wurde gewählt, da Fortran Common Blocks

keine dynamisch angelegten Arrays beinhalten dürfen. Das führte allerdings auch dazu, dass die in einer Routine benötigten globalen Arrays als Pointer übergeben werden und so die Anzahl der Aufrufargumente vieler Routinen aufblähte. Etwa:

```
real*8    xarray1(1), xarray2(1), xarray3(1)
! hier mapping der xarray1, xarray2, xarray3 auf b wie oben
…
call xmysub(xarray1, xarray2, xarray3 …)
…
subroutine xmysub (xarray1, xarray2, xarray3 …)
      real*8  xarray1(1:lenarray1)
      real*8  xarray2(1:lenarray2)
      real*8  xarray3(1:lenarray3)
      …
end subroutine
```

Ziel war, die internen Routinenaufrufe erheblich zu verschlanken und damit die Interdependenzen zwischen den MOPS-Modulen zu verringern, was einen Zwischenschritt darstellt auf dem Weg zu einer möglichst weitgehenden Loslösung der MOPS-Schnittstellenschicht vom numerischen Kern. Um das grundlegende Allokationsprinzip nicht ändern zu müssen, wurden die Mapping-Mechanismen für die Arrays leicht verändert und in zwei Include-Files zusammengefasst, die von allen Subroutinen benutzt werden können, falls diese auf globale dynamische Arrays zugreifen möchten. Dadurch müssen keine Array-Pointer mehr übergeben werden, und in allen Routinen können globale dynamische Arrays benutzt werden, als seien sie Teil des Common Blocks.

7.1.3 Callbacks

Anders als die meisten kommerziellen Solver unterstützte MOPS bis v8.19 keine Callbacks. Im Rahmen der Entwicklung des MPL-MOPS-Interfaces wurde dieses Feature als notwendige Voraussetzung der Ansteuerung von MOPS durch MPL implementiert. Ebenso benötigt MOPS Studio Callbacks, um Optimierungsläufe abbrechen zu können. Callback-Funktionen sind Routinen der Clientapplikation, die während des Optimierungsprozesses an bestimmten Stellen oder Zeitpunkten vom Solver aufgerufen werden können. Der Solver nimmt also einen „Rückruf“ in die Clientapplikation vor.
Hierzu übergibt die Clientapplikation zunächst die Adresse der aufzurufenden Callbackroutine mit der MOPS-Funktion `SetCallback`, diese wird intern gespeichert und beim Auftreten eines bestimmten Ereignisses aufgerufen. Nach Abarbeitung der Callbackfunktion kehrt die Programmsteuerung zu MOPS zurück. Ereignisse für Callbacks sind derzeit:

- Node Callbacks: Alle `n` Knoten wird der Callback ausgeführt, wobei sich die Callbackfrequenz `n` über den Parameter `xcbnnd` einstellen lässt.
- Iteration Callbacks: Im Simplex wird alle `n` Iterationen ein Callback ausgeführt. Auch hier ist `n` über den Parameter `xcbnit` frei wählbar.
- Log Callbacks: Immer wenn eine Nachricht ins Logfile geschrieben wird, wird der Log Callback aufgerufen. Die Nachricht wird auch im MOPS-Outputparameter `xlgstr` bereitgestellt (z.B. zur Bildschirmausgabe).

Eine besondere Schwierigkeit stellt die Tatsache dar, dass Fortran – anders als C/C++ – keine Funktionspointer unterstützt. Zwar können prinzipiell Funktionen im Sinne eines Pointers als Argument anderer Funktionen angegeben werden (z.B. `call Func1(Func2)`), die Speicherung von Funktionspointern in Variablen oder der Funktionspointeraufruf bei Vorhandensein von Datenargumenten sind in Fortran90 jedoch nicht oder nur auf Umwegen möglich (vgl. [McCormack06]). Um dennoch aus dem MOPS-Fortrancode ohne Einschränkungen Funktionspointer aufrufen zu können, wurde der Weg über eine kleine C++-Hilfsroutine gewählt. Funktionspointer auf Callbackfunktionen werden im MOPS Common Block als 8-Byte-Integerwerte gespeichert und durch die C++-Hilfsroutine `CBHelper()` zur Ausführung gebracht. Abbildung 35 skizziert diesen Prozess.

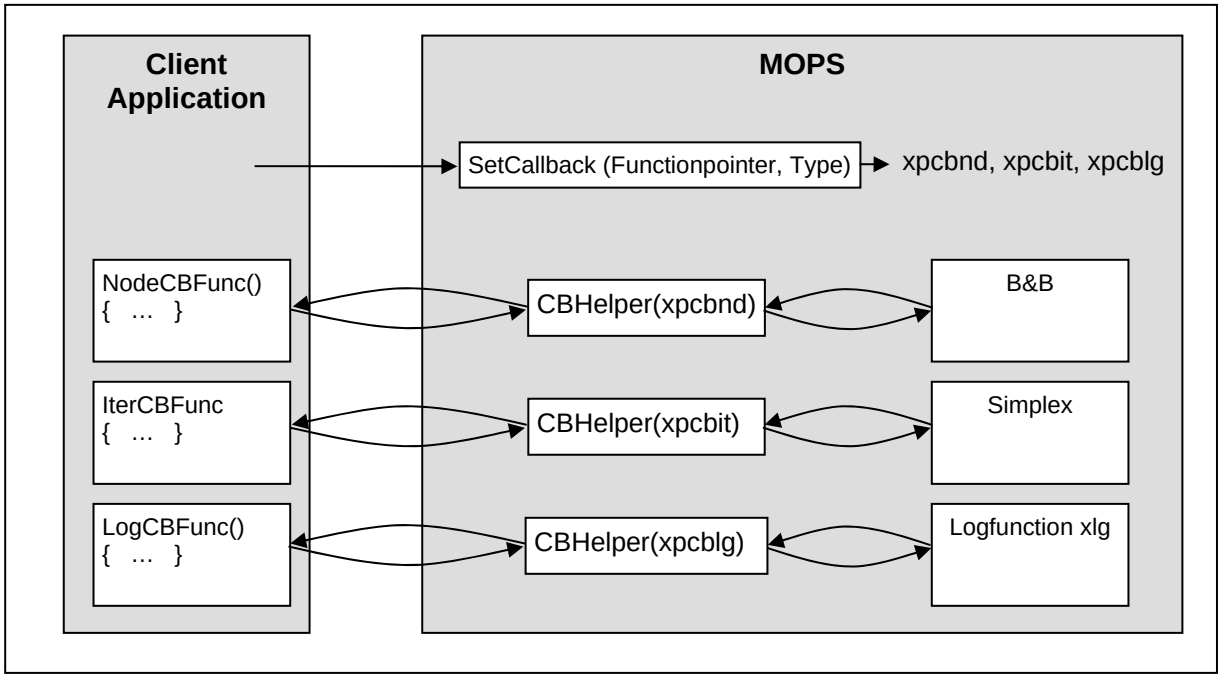


Abbildung 35: MOPS Callbacks

7.1.4 Interlanguage Interface

Um von C/C++ das Fortran-IMR-Interface bei statischer Bindung nutzen zu können, besaß MOPS schon seit frühen Versionen ein Interlanguage-Interface, das allerdings aufgrund seiner Kompliziertheit für reale Anwendungen kaum genutzt wurde. Auch hier wurde im Sinne eines Zwischenschritts zunächst ein erheblich verbessertes Interlanguage Interface entwickelt, das das Charakteristikum der direkten Zugriffsmöglichkeit auf interne MOPS-Datenstrukturen beibehielt. In einem weiteren Schritt wurde dann zusätzlich eine Lösung implementiert, die zwar keinen direkten Zugriff auf Fortran-Datenstrukturen aus C/C++ mehr erlaubt, dafür aber völlig analog zum DLL-Interface funktioniert.

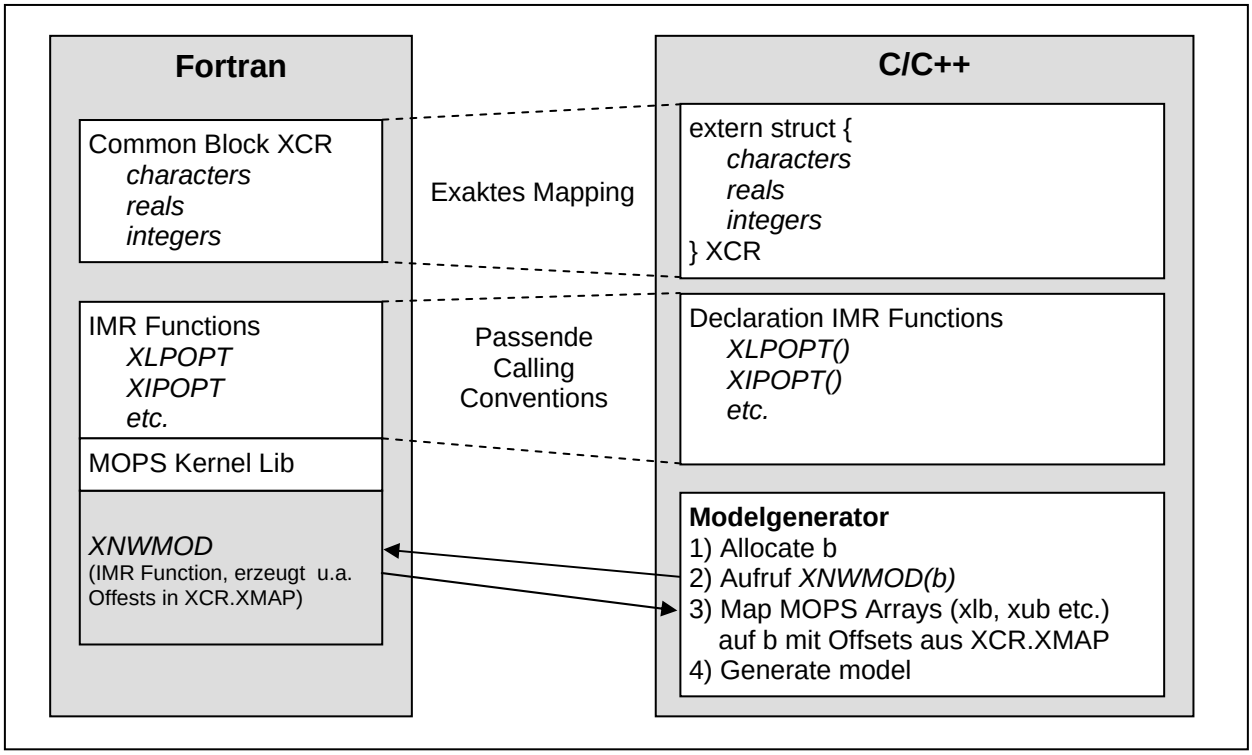


Abbildung 36: MOPS Interlanguage Interface

Grundidee des neu implementierten Interlanguage Interfaces ist es, eine C-Datenstruktur über den Fortran Common Block zu legen. Dadurch wird der Common Block mit seinen globalen MOPS-Variablen zu einem gemeinsamen Speicherbereich in C und Fortran. Der Common Block und die C-Struktur müssen sich bytegenau entsprechen, wobei nicht nur auf die exakt identische Abfolge der Variablen, sondern auch auf gleiches Alignment zu achten ist. Weiterhin müssen die Funktionen des IMR-Interfaces bezüglich Naming Conventions, Calling Conventions und Argument Passing Conventions zueinander passen. Ist das nicht der Fall, so kommt es entweder bereits beim Linking zu Fehlern, weil der Linker die Symbolnamen nicht

richtig auflösen kann, oder es kommt zu Laufzeitfehlern, weil z.B. die Calling Conventions bezüglich des Stack-Handlings nicht übereinstimmen.
Da die dynamischen Arrays, die das Modell aufnehmen (Upper/Lower Bounds, Nonzeros etc.) nicht Teil des Common Blocks, bzw. der C-Struktur `XCR` sind, müssen diese mit einem anderen Mechanismus zugänglich gemacht werden: Ein großer zusammenhängender Speicherbereich `b` wird im C-Part, dem eigentlichen Modellgenerator vorgelagert, angelegt. Danach wird ein Pointer auf `b` an die MOPS-Funktion `XNWMOD` übergeben, die die Offsets der MOPS-Arrays berechnet, diese im Array `XMAP` des Common Blocks abspeichert und anschließend zum C-Part zurückkehrt. Dort werden dann diese Offsets als Pointer benutzt, um Modelldatenarrays (z.B. `xub` für die Upper Bounds) auf den Speicherbereich b zu legen. Diese Arrays können sodann im eigentlichen C-Modellgenerator wie native C-Arrays benutzt werden, so würde z.B. `xub[0] = 123;` die Obergrenze der ersten Strukturvariablen auf 123 setzen. Die beschriebene Architektur bietet somit vollen Zugriff auf MOPS-interne Variablen und Datenstrukturen aus C/C++.
Die zweite – sehr viel trivialere – Möglichkeit, MOPS bei statischem Linking aus einem C/C++-Modellgenerator zu nutzen, besteht darin, aus dem Fortran Sourcecode eine LIB zu erzeugen, in der die exportierten Routinen bezüglich der Calling und Naming Conventions von C aufgerufen werden können. Dazu werden die MOPS-Funktionen mit der Compilerdirektive

```
!DEC$ ATTRIBUTES C, ALIAS:'_MyFunction' ::  MyFunction
```

exportiert und im C++-Part als `extern "C"` deklariert. Darüber hinaus müssen lediglich die unterschiedlichen Stringrepräsentationen in C und Fortran beachtet werden. Die so gestaltete statische Schnittstelle lässt sich exakt wie die DLL-Schnittstelle benutzen, gestattet aber keinen Zugriff auf MOPS-interne Speicherstrukturen.

7.2 COM-Komponentenbibliothek

Nachdem bereits in 6.4.3 die Komponenten der COM-Bibliothek sowie die wichtigsten Designaspekte erläutert wurden, sollen hier nun einige programmiertechnische Details der Implementation herausgegriffen werden. Zwecks besseren Verständnisses sei auch noch einmal auf die Anwendungsbeispiele der COM-Bibliothek in Kapitel 8.1 verwiesen.

7.2.1 Verwendete Technologien

Die COM-Komponentenbibliothek besteht insgesamt aus ca. 20.000 Lines of Code (mit Headern) und wurde komplett in C++ mit Visual Studio 2003/2005 erstellt. Als ausführbare Datei liegt sie in ihrer prozessinternen Version als *mocom.dll* und in der prozessexternen Version als *mocom.exe* vor. Zur COM-Programmierung wurde die *Active Template Library (ATL)* verwendet, da diese im Vergleich zu den alternativ einsetzbaren *Microsoft Foundation Classes (MFC)* kleinere und kompaktere Komponenten mit weniger Overhead erzeugt. Die MFC wurden allerdings auch in Teilen des Projekts eingesetzt, insbesondere, um die grafische Oberfläche der Property Page für die Auswahl der Solver-Parameter zu gestalten. Daneben wurde die C++ *Standard Template Library (STL)* verwendet, um die interne Modellverwaltung zu realisieren. Wie üblich bei COM-Projekten, sind die Interfaces in *Microsoft Interface Definition Language (MIDL)* formuliert und werden vom MIDL-Compiler in Ressourcen- und C-Files übersetzt. ATL und MFC sind statisch in die mocom.dll bzw. mocom.exe gelinkt, um Distributionsproblemen zu vermeiden.

Aus Gründen der Vereinfachung, und da momentan ohnehin nur ein einziger Solver angebunden ist, wurden die in Kapitel 5 erwähnten Komponenten `MOModel` und `MOSolver` zu einer einzigen Komponente `MOModel` zusammengefasst.

C++ wurde aus folgenden Gründen zur Realisierung der Bibliothek herangezogen:

- Größtmögliche Flexibilität bei der COM-Entwicklung
- Vorhandensein der mächtigen Frameworks ATL, MFC und STL
- Sehr hohe Performance
- Möglichkeit des Übergangs von COM zu .NET unter Weiterverwertung großer Teile der Codebasis bei Migration von C++ zu Managed C++

7.2.2 Interfaces der Hauptkomponente

Die Hauptkomponente der Bibliothek, `MOModel`, liegt in zwei Versionen vor: Als *OLE-Control* und als *Simple COM Object*. Tabelle 19 listet die jeweils implementierten Interfaces auf.

Interface	Bedeutung
Interfaces Simple COM-Object	
`IMOModel`	Hauptinterface. Methoden und Eigenschaften siehe Anhang
`IDispatch`	Standdardinterface für Automatisierung
`ISupportErrorInfo`	Liefert explizte Fehlermeldungen bei Exceptions
`IConnectionPointContainer`	Zur Auslösung von Events (z.B. OptimizationFinished Event)

`IProvideClassInfo`	Liefert ein ITypeInfo Interface z.B. für Typelibrarybrowsing in Entwicklungsumgebungen
`IProvideClassInfo2`	Erweiterung zu IprovideClassInfo
Zusätzliche Interfaces OLE-Control	
`IOleObject, IDataObject, IOleControl IViewObject, IViewObject2,IViewObjectEx, IOleInPlaceObject IOleInPlaceObjectWindowless, IOleInPlaceActiveObject, IQuickActivate`	Obligatorische Standardinterfaces für OLE-Controls (Rendering etc.)
`IPersist, IPersistStreamInit, IPersistStorage`	Ermöglichen Objektpersistenz des Controls, so dass Solver-Parameter und andere Eigenschaften von der Entwicklungsumgebung in einer so genannten *Property Bag* gespeichert werden können
`ISpecifyPropertyPages`	Liefert Property Pages zur Einstellung der MOPS-Parameter

Tabelle 19: Interfaces von MOModel

Der Grund für das Vorhandensein der zentralen Komponenten `MOModel` als Control und alternativ als einfaches COM Object ist, dass beide situativ unterschiedliche Vorteile haben. Ein Control hat den Vorteil, dass Eigenschaften und Property Pages zur Entwurfszeit z.B. zur Einstellung der Solver-Parameter genutzt werden können (Abbildung 37). Nachteil sind allerdings Einschränkungen beim Multithreading, die es beispielsweise nicht erlauben, ein Control in einem Backgroundworker-Thread laufen zu lassen, ebenso können Controls nicht über Prozessgrenzen gemarshaled werden. Control und Simple COM Object werden mittels einiger Preprozessordirektiven aus der gleichen Codebasis erstellt.

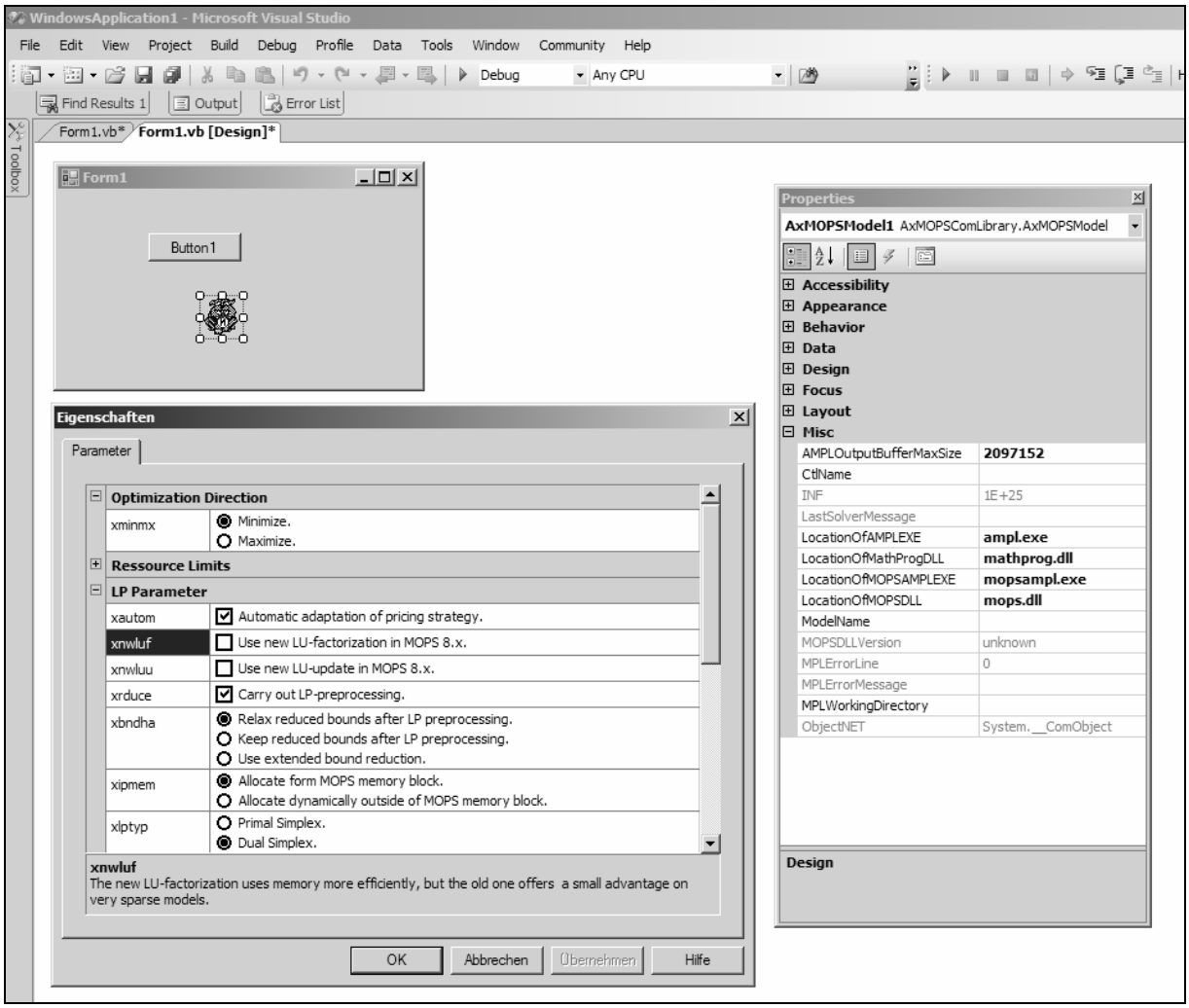


Abbildung 37: MOModel als OLE-Control

7.2.3 Interne Modellspeicherung

Die Komponente `MOModel` speichert und verwaltet das komplette Modell selbst und übergibt es erst zur Optimierung an den Solver. Bei der MOPS.DLL geschieht das mit der Funktion `PutModel`. Diese interne Modellverwaltung muss vor allem performant und dynamisch in Bezug auf die Speicherverwaltung sein. Aus Gründen der Stabilität sind objektorientierte Datenstrukturen vorzuziehen, auch wenn hierdurch ein gewisser Performanceverlust entstehen mag. Nach Tests mit diversen dynamischen Array-Klassen aus ATL, MFC und STL fiel die Wahl aus Performance- und Flexibilitätsgründen auf die STL-Templateklassen `vector`, `set`, `pair`, und `map`. Die Spalten, Zeilen und Nonzeros der Matrix werden intern als Objekte gespeichert. Hier die vereinfachte Darstellung der Deklarationen:

```
class CCol  {
      public:
            CString csName;
            MOColRowStatusType eStatus;
            double dLB, dUB, dIPSol, dLPSol, dRedCost, dCost;
```

```
            MOColType eType;
            … };

class CRow  {
      public:
            CString csName;
            MOColRowStatusType eStatus;
            double dLB, dUB, dIPSol, dLPSol, dDual;
            … };

class CNZItem  {
      public:
            long        lColIndex1;
            long        lRowIndex1;
            double      dElement;   };
```

Trotz Objektorientierung werden in der derzeitigen Implementation für ein Spaltenobjekt nur 64 Byte, für ein Zeilenobjekt 56 und für ein Nonzero 16 Byte benötigt. Ein STL-Vector nimmt die Spalten- und Zeilenobjekte auf und bildet ein dynamisch verwaltbares und wahlfrei zugreifbares Array:

```
vector <CCol> vecCol;
vector <CRow> vecRow;
```

Die Nonzeros werden hingegen in einem Objekt der STL-Templateklasse `set` abgespeichert:

```
set <CNZItem, CNZItemCompare> NZItemSet;
```

Ein STL-`set` ist ein assoziativer Container variabler Größe, der eine geordnete Menge von unikalen Elementen beinhaltet (vgl. [Stroustrup97, Kap. 17]). Sets sind optimiert für Such-, Einfüge- und Löschoperationen, wobei der Zeitaufwand für die genannten Operationen durchschnittlich proportional zum Logarithmus der Anzahl der Elemente im `set` ist (vgl. [Microsoft07h]). Sets können von entsprechenden `set`-Iteratoren bidirektional durchlaufen werden. Die Ordnung eines Sets wird von einer so genannten *Traits-Funktion* bestimmt, die als Templateparameter in die Instanzierung des `set`-Objekts eingeht. Eine Traits-Funktion liefert *Wahr* oder *Falsch* als Ergebnis des Ordnungsvergleichs zweier Argumente („Binärprädikat“). In `MOModel` erzeugt die Traits-Funktion `CNZItemCompare` standardmäßig eine zeilenweise Anordnung der Nonzeros. Durch Austausch der Traits-Funktion kann bei Bedarf auch eine spaltenweise Anordnung der Nonzeros erzeugt werden, was z.B. beim Schreiben von MPS-Files genutzt wird.

Für schnelle Suchoperationen werden die Spalten- und Zeilennamen zusätzlich in `map`-Objekten gespeichert, die deklariert werden als:

```
map<CString, long>  MColNames;
map<CString, long>  MRowNames;
```

Dabei werden die Spalten- und Zeilennamen einem Index zugeordnet, der die Position innerhalb des Spalten- bzw. Zeilen-Vektors (`vecCol`, `vecRow`) angibt. Ein STL-`map` ist eine Klasse, die einem eindeutigen Schlüssel (hier dem Namen) einen Wert (hier den Index) zuordnet. Maps verwenden eine dynamische Speicherverwaltung und müssen daher nicht preallokiert zu werden. Die Schlüssel werden als balancierter Binärbaum (Red-Black-Tree) gespeichert, was den Vorteil hat, dass Such- Einfüge- und Löschoperationen logarithmische Komplexität aufweisen ([Josuttis99, S. 157 f.]). Die Suche nach Spalten- und Zeilennamen in einer Map mit 1.000 Elementen, ist daher im Mittel rund 50-mal schneller als eine einfache sequenzielle Suche. Hier wäre allerdings noch Potenzial für Verbesserungen in Bezug auf die Performance, denn eine Suche über Hash-Tables ist für gewöhnlich ca. fünf- bis zehnmal schneller als die Binärbaum-Suche ([Josuttis99, S. 199 f.]). Unterstützung für Hash-Tables, bietet der Namensraum `stdext` mit den Templateklassen `hash_set` und `hash_map`, deren Verwendung in der Komponentenbibliothek noch auszutesten ist.

7.2.4 Handling von VARIANTS

Wie in 6.4.3 beschrieben, macht die COM-Bibliothek bei Methoden und Eigenschaften sehr häufig Gebrauch von `VARIANT`-Argumenten, um die fehlende Möglichkeit der Funktionsüberladung nachzubilden. Intern werden alle skalaren `VARIANT`s – falls möglich – in folgende Grundtypen umgeformt: `DOUBLE`, `LONG`, `BSTR` und Interface-Pointer vom Typ `IDispatch*`. Beinhaltet der `VARIANT` ein `SafeArray`, so wird dieses, falls es in der Komponente gespeichert werden muss, in die intern verwendeten STL-Container (meist vom Typ `vector`) übertragen. Zur Erleichterung dieser Konvertierungen wurden diverse Hilfsklassen erstellt.

`VARIANT`s finden weiterhin ausgiebige Verwendung bei Eigenschaftsaufrufen, die die Angabe von Indexwerten als Eigenschaftsparameter benötigen, wobei immer folgendes Schema benutzt wurde:

```
[id(DISPID_VALUE), propget] HRESULT Value(
      [in, optional] VARIANT Index1, [in, optional] VARIANT Index2,
      [in, optional] VARIANT Index3, [in, optional] VARIANT Index4,
      [in, optional] VARIANT Index5, [out, retval] VARIANT* pVal);
```

Diese Formulierung in MIDL erlaubt die Abfrage eines Eigenschaftswerts vom Typ `VARIANT`, wobei optional bis zu fünf Indexwerte angegeben werden können. So würde beispielsweise auf eine Zelle eines zweidimensionalen, über Jahre und Monate indizierten `MODataArrays` in VB zugegriffen mit:

```
Dim a as new MODataArray
```

```
Dim v as new Object
…
v = a(2007,”Jan”)    'Value kann weggelassen werden, weil default
```

Mit dem Attribut [in, optional] versehene VARIANTs sind der einzige Weg, um für eine Eigenschaft eines COM-Objekts optionale Eigenschaftsparameter anzugeben. Die Verwendung von SafeArrays vom Typ Variant, wie sie bei Methoden mit variabler Anzahl von Argumenten eingesetzt wird, scheidet für Eigenschaften aus. Dies liegt daran, dass das letzte Element der Argumentliste einer Eigenschaft immer der Eigenschaftswert selbst sein muss, der entweder als [in] oder als [out] gekennzeichnet wird, je nachdem ob die Eigenschaft ausgelesen oder gesetzt wird. Dennoch sind bei der oben genannten Syntax die maximal benutzbaren Parameter nicht auf fünf begrenzt, denn dank der VARIANTs kann auch ein COM-Interface (IDispatch*) als Parameter angegeben werden, was bei folgendem Beispiel ausgenutzt wird:

```
Dim a as new MODataArray
Dim v as new Object
Dim t as new MOTuple
…       ' a wird 6-dimensional initilisiert
v = a(t.Set(Index1,Index2,Index3,Index4,Index5,Index6))
```

Das Hilfsobjekt MOTuple t wird über Set(…) mit sechs Indexwerten initialisiert. Set(…) ist aber nicht als Methode, sondern als Eigenschaft implementiert und liefert nach Initialisierung eine Referenz auf sich selbst. Diese geht als erster Parameter in die parametrisierte Value-Eigenschaft von a ein, worauf diese den Inhalt des so spezifizierten Feldes liefert. Das Beispiel zeigt, dass sich durch geschickte Verwendung von VARIANTs die meisten syntaktischen Beschränkungen von COM unter VB umgehen lassen.

7.2.5 Interaktion zwischen Komponenten

Die Komponentenbibliothek ist so aufgebaut, dass alle modellbezogenen Informationen in der zentralen Komponente MOModel enthalten sind und es keine verteilten oder doppelt vorhandenen Datenbestände gibt. Sobald ein Modell einmal generiert ist, können hierzu benutzte Hilfskomponenten, wie z.B. MODataArrays oder MOVarSets, gelöscht werden, ohne dass das Modell Informationen verlieren würde. Die Komponenten MOVar, MOVarSet, MORes und MOResSet beinhalten nur Referenzen auf die entsprechenden Variablen oder Restriktionen im MOModel, und nur dort sind Werte wie UB, LB, RHS etc. gespeichert. Umgekehrt enthält das MOModel Pointer auf die referenzierenden Komponenten, so dass eine bidirektionale Verbindung entsteht. Abbildung 38 stellt dieses Prinzip dar und zeigt eine

MOVar-Komponente, die einen Indexpointer auf die entsprechende Position im Column-Vector der MOModel-Komponente besitzt. Wird nun beispielsweise der UB dieses Variablenobjekts geändert, so erfolgt die Änderung, für den Benutzer transparent, im MOModel-Objekt selbst. Die Kommunikation erfolgt dabei ausschließlich über die Interfaces der Komponenten, es gibt also keine direkten Speicherpointer wie in C++. Durch Löschungen oder Einfügungen im Colum-Vector können sich diese Indexreferenzen ändern, so dass das MOVar-Objekt diese Änderung nachvollziehen muss. Hierzu besitzt das MOModel-Objekt eine Multimap-Tabelle, die Interfacepointer auf alle aktuell verbundenen MOPSVar-Objekte enthält. Diese Interfacepointer dienten zur Benachrichtigung der von der Indexreferenzänderung betroffenen Variablenobjekte. Für MOVarSet, MORes und MOResSet existieren analoge Mechanismen.

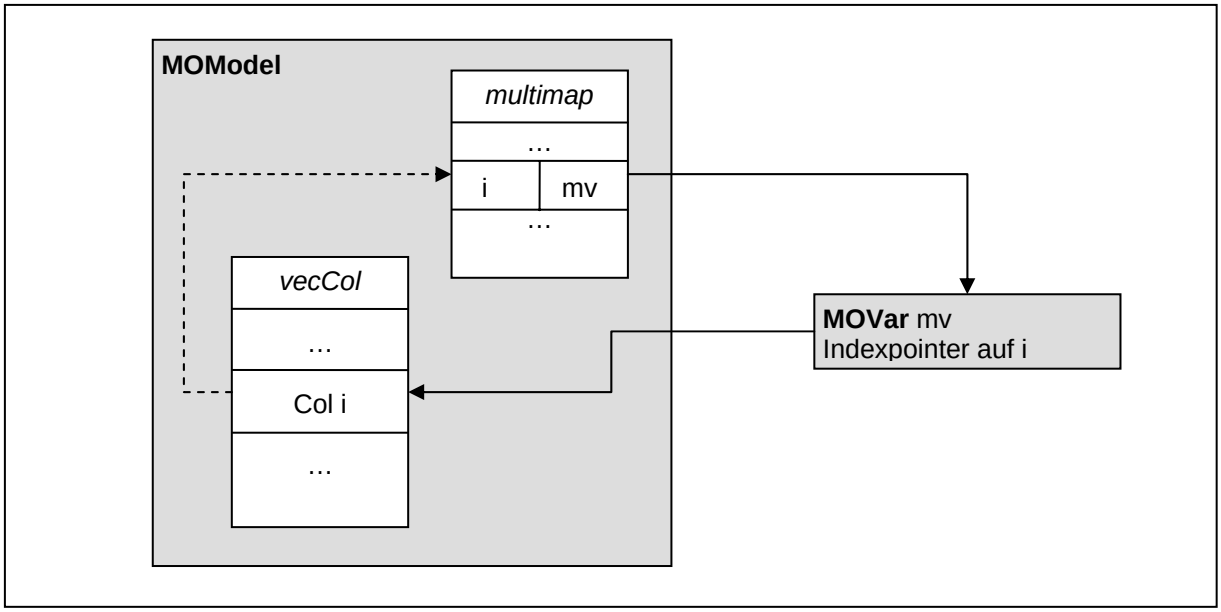


Abbildung 38: Komponentenreferenzen

7.3 Modellierungssprachenschnittstellen

Schnittstellen zu den vier Modellierungssprachen AMPL, MathProg, Lindo und MPL wurden zunächst im Rahmen der Komponentenbibliothek implementiert und werden über diese ebenfalls in MOPS Studio verwendet.

7.3.1 AMPL

Die Komponente MOModel enthält eine Methode

```
ReadAMPLModel (ModFileName, DatFileName1, …)
```

die den in Abbildung 39 skizzierten Prozess startet: Zunächst wird ein temporäres *AMPL Command File* generiert, das Anweisungen für das interaktive Kommandozeileninterface der AMPL.EXE enthält. Das Command File gibt vor allem die Speicherorte der .mod- und .dat-Dateien an und setzt außerdem diverse Optionen. Anschließend wird der AMPL.EXE-Prozess erzeugt und als Kommandozeilenparameter der Name des AMPL Command Files mitgegeben. Zur Kommunikation mit dem Prozess wird eine Pipe eingerichtet, auf die die I/O- und Error-Channels des Prozesses gelenkt werden. AMPL führt das Command File aus und schickt seine Status- und Fehlermeldungen über die Pipe an die Komponente `MOModel`. Bei erfolgreicher Verarbeitung der Modell- und Datenfiles legt AMPL so genannte Stub-Files an, die von der `MOModel`−Komponente gelesen werden können und aus denen diese das generierte Modell extrahiert. Das Auslesen der Stub-Files geschieht mithilfe der AMPL-Bibliothek *amplsolv.lib*, die den binären Inhalt der Stub-Files in eine komplexe Datenstruktur (`ASL`) einliest (Details in [Gay97]). Von dort muss das Modell in das interne Format der `MOModel`-Komponente umgeformt werden. Schließlich werden der AMPL-Prozess beendet und temporäre Dateien gelöscht.

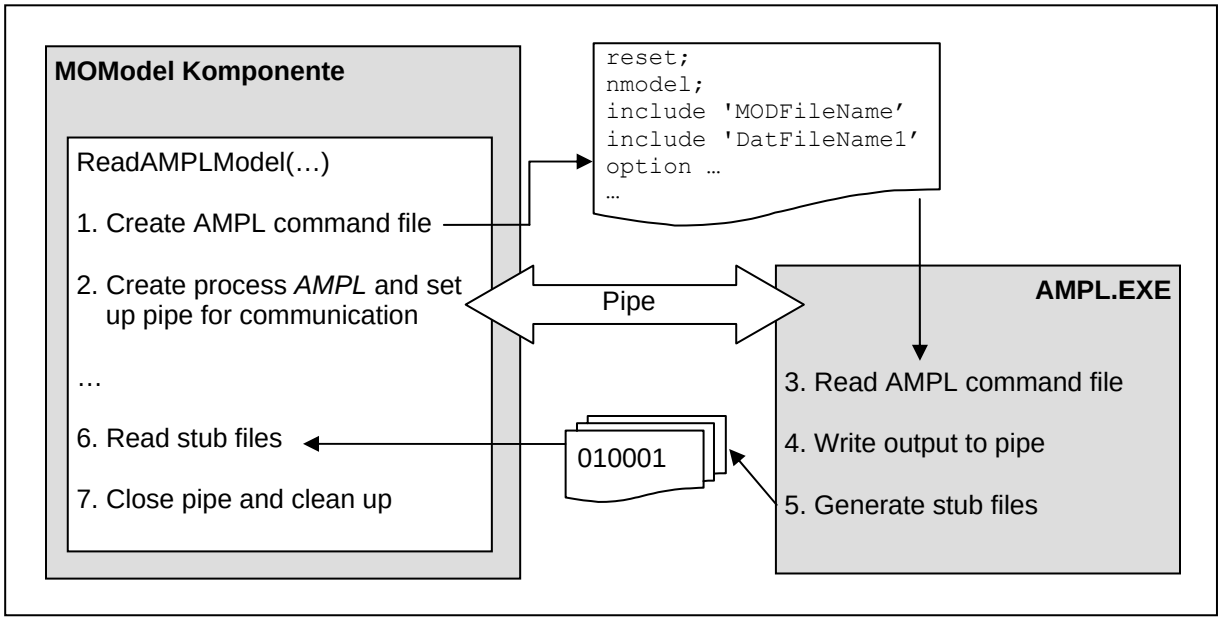


Abbildung 39: AMPL-Schnittstelle

Falls das AMPL-Modell jedoch selbst `solve`-Statements enthält, muss ein anders Schema gewählt werden, da AMPL nur Solver ansprechen kann, die als ausführbare Datei vorliegen. Diese Beschränkung hängt mit der reinen Kommandozeilenorientierung von AMPL zusammen. Die Komponente `MOModel` enthält für solche Modelle die Methode

```
ExecuteAMPLModel (ModFileName)
```

die auch z.B. von MOPS Studio für das gleichnamige Feature angesprochen wird. Der Mechanismus dieser Methode beginnt wie bei „normalen“ AMPL-Modellen mit der Generierung eines AMPL Command Files, dem Starten des AMPL-Prozesses und der Einrichtung einer Pipe zu diesem Prozess. Trifft nun AMPL auf den `solve`-Befehl, so werden die binären Stub-Files erzeugt. Der anschließend aufzurufende Solver muss aber in Form einer ausführbaren Datei vorliegen. Hierzu dient der Proxy-Prozess MOAMPL.EXE, der den Aufruf entgegennimmt und dann einige versteckte, weil nur für den Proxy relevante, Methoden der `MOModel`-Komponente aufruft: `ReadStubFile` liest die Stubs ein und überträgt sie in das interne Modellformat, anschließend wird optimiert und im Erfolgsfall mit `WriteSolStub` die Lösung als Solution-Stub-File geschrieben. Danach wird MOAMPL.EXE beendet, und AMPL übernimmt wieder die Steuerung, indem es den Solution-Stub einliest. Für gewöhnlich erfolgt dann eine Weiterverarbeitung der Lösungsergebnisse in AMPL. Häufig werden auch mehrere Lösungsläufe iterativ hintereinander ausgeführt.

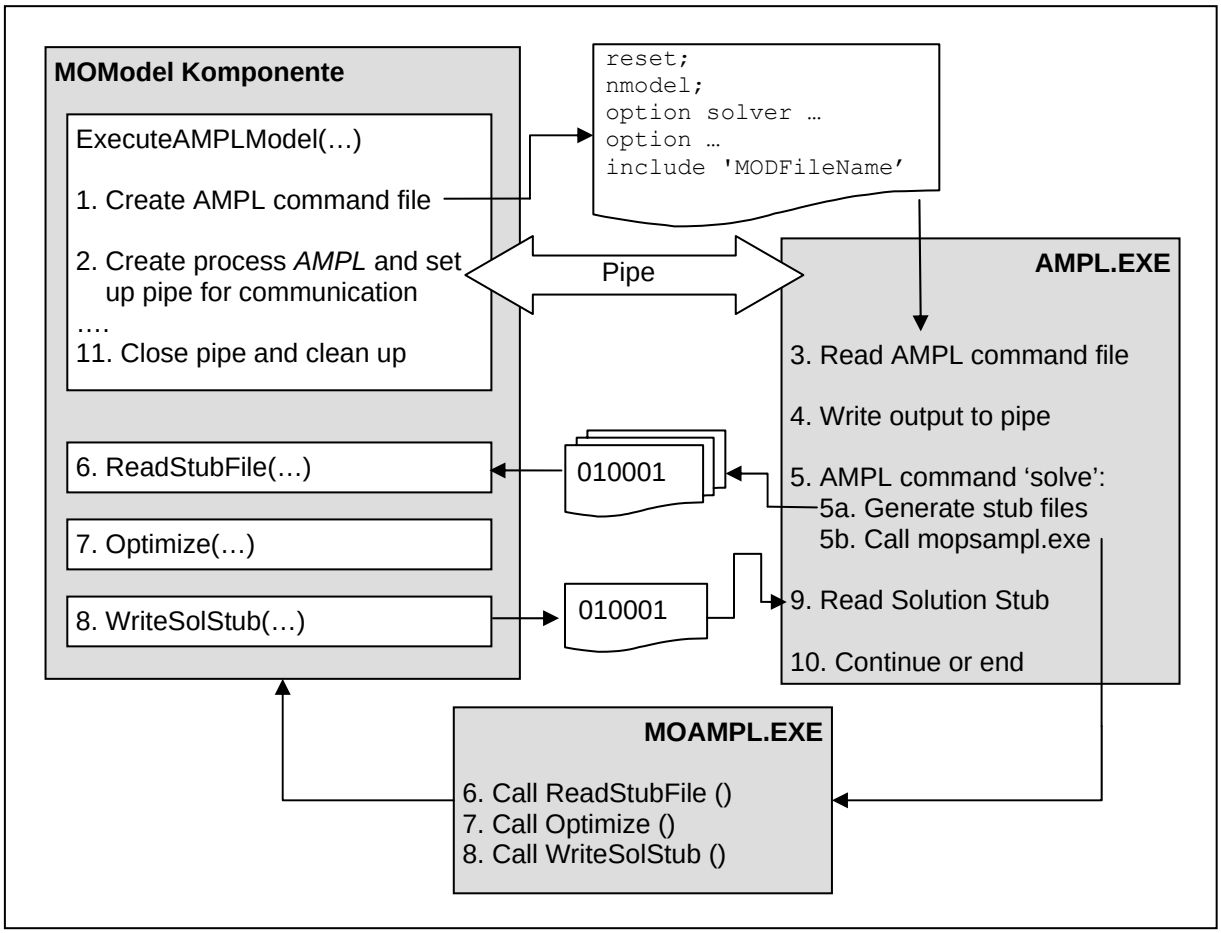


Abbildung 40: Execute AMPL

7.3.2 GNU MathProg

Im Vergleich zu AMPL ist die MathProg-Schnittstelle recht einfach zu implementieren. Dazu wurde zunächst um die rund 100 Funktionen der GNU MathProg Library ([GNU06]) eine Kapsel in Form einer DLL gelegt, die die Benutzung der Bibliothek vereinfacht, indem sie die Notwendigkeit der Angabe von Modellpointern (`LPX*`) eliminiert. Die so erzeugte MathProg.dll ist unabhängig von anderen Bibliotheken und kann leicht distribuiert werden. Der entsprechende Programmcode ist sehr simpel und völlig unabhängig von der Middleware-Bibliothek. Er wurde als Open-Source veröffentlicht und kann auch für andere Zwecke benutzt werden.

Die Komponente `MOModel` besitzt zum Lesen von MathProg-Modellen die Methode `ReadMathProgModel(MODFileName, DATFileName1, …)`. MathProg kann eigentlich nur eine einzige Datendatei verarbeiten, die Methode erlaubt aber, um Analogie zu AMPL herzustellen, die Angabe mehrerer Daten-Files, die intern zusammengefügt werden. Die `MOModel`-Komponente ruft intern die von der MathProg.dll exportierten Methoden auf, um zunächst das Modell von der Festplatte einzulesen (`lpx_read_model`) und es anschließend über die diversen Modellabfragefunktionen von MathProg (z.B. `lpx_get_col_ub, lpx_get_col_lb`) in das komponenteninterne Format zu übertragen. MathProg-Fehlermeldungen und sonstige Statusausgaben landen in einem gesonderten Outputfile, das ebenfalls gelesen und ausgewertet wird. Bei Fehlern im Modell liefert die Komponente einen entsprechenden `HRESULT`-Fehlercode, der in Umgebungen wie VB eine Exception auslöst.

7.3.3 LINDO

Die sehr einfache Modellierungssprache LINDO, die eher ein Dateiformat als eine Modellierungssprache ist, kann von der `MOModel`-Komponente gelesen und geschrieben werden. Zum Einlesen von LINDO-Modellen besitzt die Komponente einen integrierten Parser, der aus ca. 400 Zeilen C++-Code besteht, auf den aber hier nicht weiter eingegangen werden soll.

7.3.4 MPL

Die COM-Komponente `MOModel` hat eine Methode `ReadMPLModel (FileName)`, die MPL als Server benutzt und ein MPL-Modell einliest. Dazu wird anfangs über die Registry das Vorhandensein der Optimax-Bibliothek überprüft. Die COM-Bibliothek Optimax enthält eine Reihe von Komponenten zur Modellierung und hat gewisse Ähnlichkeit mit der hier

entwickelten COM-Bibliothek. Aus der Optimax-Bibliothek werden zunächst ein `mplOptimax`- und ein `mplModel`-Objekt instanziert, und die Modelldatei wird in das `mplModel`-Objekt geladen. Falls dies erfolgreich war, wird ein `mplMatrix`-Objekt aus dem `mplModel` instanziert. Aus diesem `mplMatrix`-Objekt können sodann unter Zuhilfenahme weiterer MPL-Komponenten (`mplVariables` etc.) die Modelldaten extrahiert und in das interne Format der `MOModel`-Komponente übertragen werden.

Für den Optimierer MOPS existiert ebenfalls eine eigene MPL-MOPS-Schnittstelle, die zur Einbindung der MOPS.DLL in MPL und MPL Studio dient. Bei dieser Schnittstelle wird MOPS von MPL als Optimierungsserver benutzt. Das Interface besteht auf MPL-Seite aus dem (nicht öffentlichen) C-Code des MPL-Interfacemoduls *mplmops.c*. Dieses Modul stellt an den anzusteuernden Solver bestimmte Anforderungen bezüglich notwendiger Features. Diese neuen Features wurden zwar primär für MPL in MOPS implementiert, da die hier entwickelte Komponentenbibliothek aber auch einige der Features nutzt, sollen diese kurz beschrieben werden.

Neben den bereits erwähnten Callbacks mussten vor allem die Parameterzugriffsfunktionen der MOPS.DLL überarbeitet werden, so dass sie dem Schema entsprechen, das die meisten anderen Solversysteme in ihren Callable Libraries verwenden (siehe Vergleich in Anhang 1). Dieses Schema sieht unterschiedliche Funktionen für den Zugriff auf Fließkomma-, Integer- und String-Parameter vor und erfordert insbesondere für C/C++ die Definition von symbolischen Konstanten für die Parameter-IDs. Daher wurden die bisherigen Parameterzugriffsfunktionen `GetParameter` und `SetParameter` ergänzt um folgende DLL-Funktionen:

```
GetIntParameter(ID,[out]IntVal) / SetIntParameter(ID, [in]IntVal)
GetDblParameter(ID,[out]DblVal) / SetDblParameter(ID, [in]DblVal)
GetStrParameter(ID,[out]StrVal) / SetStrParameter(ID, [in]StrVal)
```

Weiterhin wurden Funktionen zur Abfrage von Min-, Max- und Default-Werten von Parametern erforderlich:

```
GetIntParamInfo(ID,[out]Name,[out]IntVal,[out]Default,[out]Min,[out]Max)
GetDblParamInfo(ID,[out]Name,[out]DblVal,[out]Default,[out]Min,[out]Max)
```

Diese eigentlich recht einfachen Ergänzungen machten allerdings intern ein Redesign der Datenstrukturen für die Parameterspeicherung erforderlich: Waren die Parameter bisher als einfache, globale Variablen im Fortran Common Block gespeichert, so mussten nun Arrays von Strukturen eingeführt werden. Diese Struktur speichert z. B. einen Integer-Parameter:

```
TYPE TIParaStruct
    SEQUENCE
      integer*4 min
      integer*4 max
```

```
      integer*4 def
      integer*4, pointer :: pvar
      integer*4 iotyp    ! 0=Input parameter, 1=Output parameter
      character*8 name
END TYPE
```

Damit die bisherigen Variablen des Common Blocks beibehalten werden können, hält die Struktur einen Pointer auf diese Variablen, so dass z.B. `pvar` auf `xbrheu` (Parameter zur Steuerung der Branching-Heuristik) zeigt.
Die symbolischen Konstanten der rund 130 Parameter-IDs werden für C/C++ in einem Header definiert, dessen Einträge folgende Form haben:

```
// Number of parameters
#define MOPS_INT_PARAME TER_COUNT  77
…
// Integer parameter (input/output)
#define MOPS_INT_XADROW             1
#define MOPS_INT_XAUTOM             2
…
```

Die MOPS.DLL besitzt die Funktion `WriteParamHeader(FileName)`, mit der ein passender Parameter-Header erzeugt werden kann.

7.4 MOPS Studio

Beim Integrierten Modellierungssystem *MOPS Studio* handelt es sich um eine weitgehend in Visual Basic .NET entwickelte Applikation, die auf der hier beschriebenen und implementierten COM-Komponentenbibliothek aufsetzt. Dabei wurde in den ersten Versionen (im Jahr 2005 und 2006) das .NET-Framework 1.1 verwendet und im Jahr 2007, vor allem aus Kompatibilitätsgründen zu *Vista*, auf das .NET-Framework 2.0 migriert. MOPS Studio besitzt ein Multi-Document-Interface (MDI), das das parallele Bearbeiten und Optimieren von mehreren Modellen erlaubt. Tabelle 20 beschreibt die verwendeten Fensterklassen (*Windows Forms*).

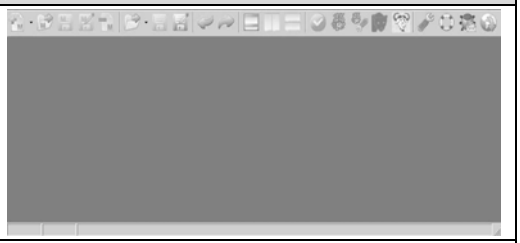
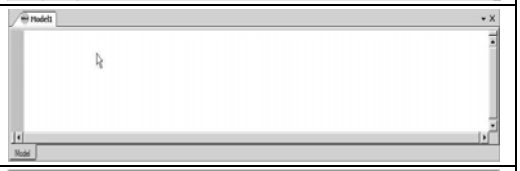
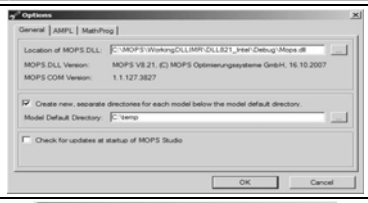
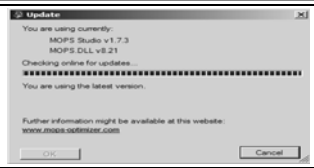
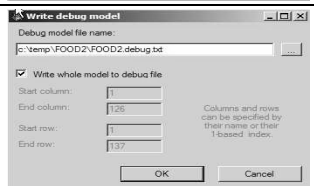
Form	Funktion	Aussehen
`MainForm`	Hauptfenster und MDI-Container für alle anderen Forms sowie für Symbol- und Statusleisten	
`ModelForm`	MDI-Child mit umfangreichen Docking-Funktionalitäten. Zeigt die Modelldateien auf unterschiedlichen Tabs	
`OutputForm`	Dockbares Fenster mit Auto-Hide-Funktion für alle Statusausgaben	
`OptionsForm`	Fenster zur Einstellung diverser Optionen wie z.B. zu verwendende MOPS.DLL, MathProg.DLL oder AMPL.EXE	
`UpdateForm`	Dialogfenster für Auto-Updates über das Internet. Anhand einer vom Server bezogenen XML-Datei wird überprüft, ob die vorliegende Version noch aktuell ist.	
`DebugModelForm`	Dialogfenster, in dem Filename und Modellausschnitt eines zu schreibenden Debug-Modells angegeben werden können	
`DebugFileForm`	Fenster zur Anzeige von Debug-Files	ohne Abb.
`ReportBugForm`	Kontakthinweis zum Entwickler-Team zur Meldung von Bugs	ohne Abb.
`AboutForm`	Copyright Hinweis	ohne Abb.

Tabelle 20: MOPS Studio Form-Klassen

Neben den aufgeführten Forms enthält MOPS Studio weitere .NET-Komponentenklassen, die zum Teil grafische Oberflächen haben, weil sie von grafschen Controls abgeleitet wurden. GUI-bezogener Code für grafische Darstellung und Benutzerinteraktion liegt dabei naturgemäß eher in den Forms, während sich der übrige der Code (z.B. die Aufrufe der Modellierungsbibliothek) in den .NET-Komponentenklassen befindet. Diese Trennung ist allerdings aus praktischen Gründen nicht scharf, sondern eher graduell. Auf eine explizite Darstellung als UML-Diagramm soll zugunsten einer einfachen Auflistung der wichtigsten .NET-Klassen mit kurzen Erläuterungen (Tabelle 21) verzichtet werden.

Klasse / Modul	Funktion
`CFileItem`	Kapselt eine Datei (z.B. MOD-, DAT-, Statistik-File etc.) mit einer Editorkomponenteninstanz und dem Tab, auf dem sich diese befindet `CFileItem` ist daher von `MopsTabPage` abgeleitet und besitzt unter anderem Methoden wie `LoadFile()` oder `SaveFile()`, um Dateien in den Editor zu laden oder daraus zu schreiben. Die Klasse steuert auch das unterschiedliche Syntax-Coloring, je nach geladenem Dateityp (AMPL, MPL etc.). Außerdem implementiert sie einen Watcher-Mechanismus, der extern vorgenommene Änderungen an der überwachten Datei registriert.
`CModel`	`CModel` ist zusammen mit `CFileItem` die wichtigste Klasse und fasst mehrere `FileItems` (etwa für MOD-, DAT-, IP- oder LP-Lösung etc.) zu einem Modell zusammen. Ein Modell enthält darüber hinaus auch Profiles und modellbezogene Einstellungen. Alles zusammen wird von `CModel` als Modell-Projekt geladen und gespeichert. `CModel` stößt auch Optimierungen an, die von der COM-Komponente `MOModel` und der MOPS.DLL durchgeführt werden, und erstellt nach Optimierung die Solution Summary. Auch das Threading während der Optimierung wird von `CModel` maßgeblich unterstützt: Zwar enthält die COM-Komponente `MOModel` bereits eine eigene Multithreading-Steuerung, `CModel` muss aber in einem Backgroundworker-Thread Fortschrittswerte, wie z.B. die Anzahl der untersuchten Knoten, von der Komponente abfragen und diese an die GUI weiterleiten
`Helper`	Dieses Modul enthält zentral für alle anderen Klassen und Forms eine Reihe von statischen Hilfsroutinen bereit, wie z.B. für das Öffnen von Lade- und Speicherdialogen mit angepassten Default-Dateinamen und –Dateiendungen oder auch Hilfsroutinen für die MDI-Steuerung.
`MopsTabControl`	Ist von `TabControl`, das zum .NET-Standard gehört, abgeleitet und erweitert dieses z.B. für Drag-and-Drop und andere spezifische Funktionalitäten.
`MopsTabPage`	Stellt eine einzelne Seite eines `MOPSTabContols` dar und ist von `System.Windows.Forms.TabPage` abgeleitet. Die Erweiterungen steuern Drag-and-Drop und Sichtbarmachung (Bei LP-Modellen ist z.B. die `TabPage` für die IP-Solution nicht sichtbar.)
`ProgressValuesTabPage`	Ist von MopsTabPage abgeleitet und stellt die Fortschrittswerte (z.B. bester bisheriger Zielfunktionswert etc.) während der Optimierung dar

Tabelle 21: MOPS Studio-Klassen

Neben den oben aufgeführten Klassen werden noch weitere Komponenten von Fremdherstellern verwendet, hierbei sei vor allem die Editor-Komponente *Editor.NET* von *Quantum Whale* (www.qwhale.com) erwähnt, die seit MOPS Studio v1.8 eingesetzt wird. *Editor.NET*

zeichnet sich besonders für die Unterstützung großer Dateien, Funktionen für Syntax-Coloring und reinen managed .NET-Code (wichtig für 64-Bit-Versionen) aus.

Fast alle modellbezogenen Funktionalitäten, wie etwa das Einlesen von AMPL- oder MPL-Modellen, die Optimierung oder die Konvertierung zwischen Modellformaten erfolgen in der Komponenten-Bibliothek, die über die .NET-COM Interoperabilitätsmechanismen (s. 4.3.5) eingebunden ist. Abbildung 40 stellt die Architektur der Implementation von MOPS Studio in der derzeitigen Version 1.8 zusammenfassend dar.

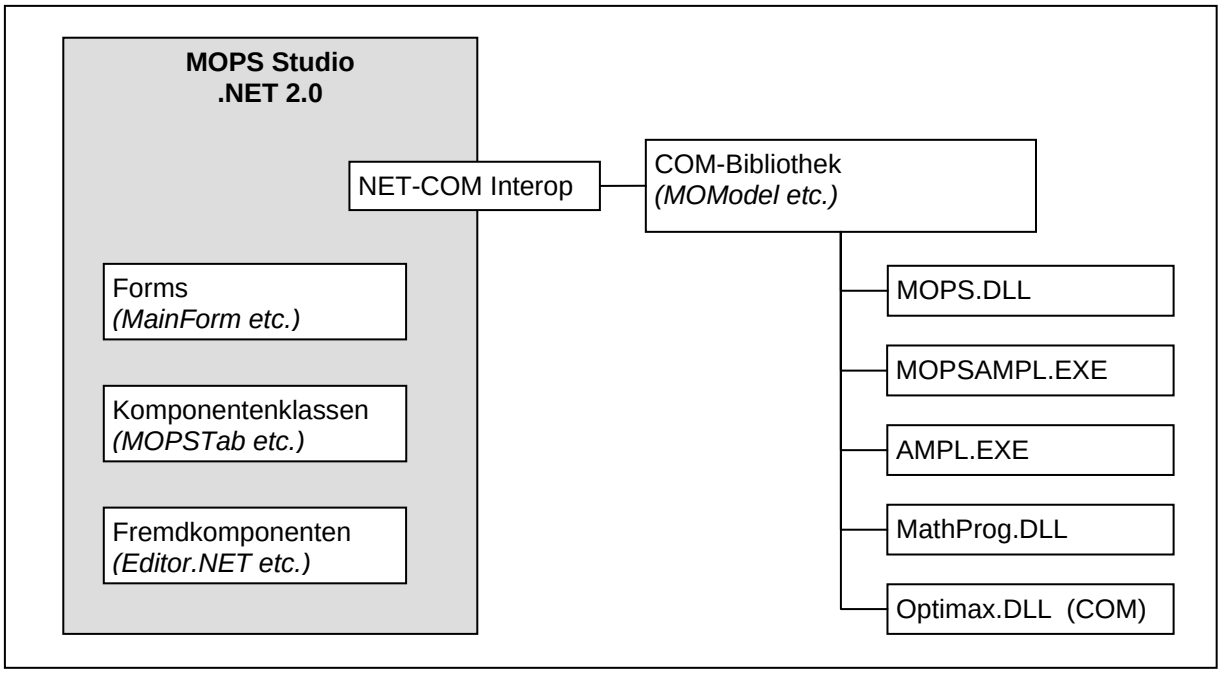


Abbildung 41: Architektur MOPS Studio

Zu MOPS Studio gehört weiterhin eine komplett neu erstellte Hilfe, die vor allem Erläuterungen zu den MOPS-Parametern und MOPS Studio-Funktionen bringt, aber auch Hilfethemen zur MOPS.DLL und MOPS.LIB und zu Dateiformaten (MPS, Triplet etc.) enthält. Teilweise ist MOPS Studio kontextsensitiv mit der Hilfe verbunden. Weiterhin wurde für MOPS Studio ein knapp 30-seitiges Tutorial „AMPL und MOPS Studio“ erstellt, das ebenfalls zur Distribution gehört.

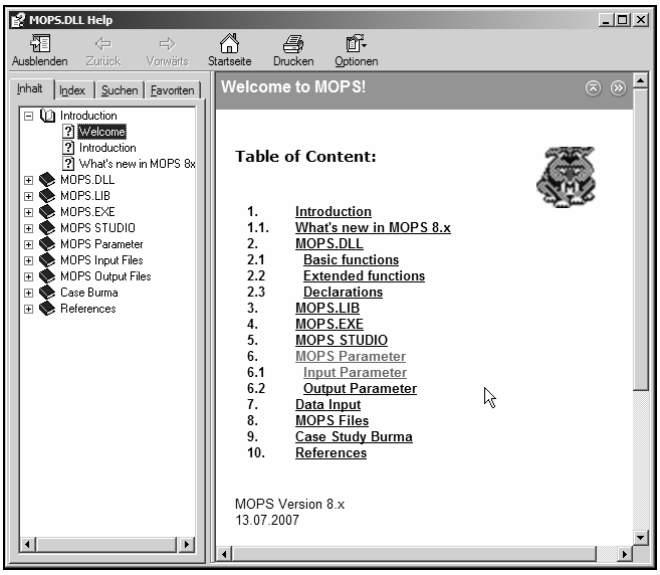


Abbildung 42: MOPS Studio-Hilfe

MOPS Studio wird als Installationspaket (.msi und .exe) für Windows 2000, XP (32/64 Bit) und Vista distribuiert. Zunächst basierte dieser Installer auf einem Visual Studio 2003 Setup-Projekt, wurde aber später wegen der gestiegenen Anforderungen an die Installationskomplexität (z.B. Seriennummer-Passwort, Prüfung auf Vorversionen, Registry Keys) mithilfe von InstallShield erstellt.