5 Kritikpunkte und Anforderungen

Obwohl die Schnittstellen vieler Optimierungssysteme schon längere Entwicklungs- und Erprobungszeiten hinter sich haben, und obwohl viele EUS-Entwickler teilweise jahrelange Erfahrungen vorweisen können, ist die Integration von Optimierungssoftware in EUS, vor allem in größeren Projekten, noch immer kein triviales, sondern ein mitunter recht fehleranfälliges Unterfangen. Im Folgenden sollen, ausgehend von einem realen Entwicklungsprojekt, die wichtigsten Problemfelder aufgedeckt werden, wobei ein besonderer Fokus auf dem Optimierungssystem MOPS liegt.

5.1 Fallstudie: Projekt EPOS

Im Jahr 1998 startete ein großer deutscher Ferngasversorger das Projekt *EPOS*, in dem der Autor dieser Arbeit in verschiedenen Funktionen tätig war. Ziel von EPOS war die Entwicklung eines integrierten Planungssystems für die Gaseinkaufsplanung. Das System wurde nach Fertigstellung der ersten Versionen (1999/2000) in den Folgejahren auf weitere Fragestellungen[1] ausgedehnt und ist momentan noch immer erfolgreich im Einsatz.

Die Entscheidung zur Entwicklung von EPOS hatte zwei Hintergründe – einen ökonomischen und einen technischen bzw. softwarestrategischen: Der ökonomische war die sich abzeichnende Liberalisierung auf den europäischen Gasmärkten, die eine erhebliche Flexibilisierung der Lieferbeziehungen mit sich bringt, was eine im selben Maße flexible Berechnung und Optimierung der Gaseinkaufsplanung erforderlich macht. Der softwarestrategische war die Ablösung des bestehenden, großrechnerbasierten Gaseinkaufsoptimierungssystems zugunsten eines modernen Systems auf PC-Basis.

Bei der Gaseinkaufsplanung geht es darum, den Bezug von Gas bei Lieferanten (vor allem Russland, Norwegen und Niederlande) kostenminimal zu gestalten. Abbildung 22 zeigt eine vereinfachte Darstellung des deutschen Ferngaspipelinenetzes mit den Gaseinspeisepunkten an den Landesgrenzen. Ausgehend von einem über stochastische Modelle (Temperatur-/Wetterverläufe etc.) festgelegten prognostizierten Gasverbrauch sind Gasmengen an den Einspeisepunkten zu entnehmen, so dass die Gesamtbeschaffungskosten minimal sind. Planungshorizont ist häufig ein Gaswirtschaftsjahr (Okt.-Sept.) oder Teile davon. Der Gasbezug unterliegt jedoch einer Reihe von Restriktionen, die einzuhalten sind:

---

[1] So wurde z.B. im Jahr 2004 im Projekt EPOS-BO die Einbeziehung von Gas-Druck-Mengenverhältnissen auf die Einkaufsplanung und für andere Bereiche untersucht. Auch an diesem Projekt war der Autor beteiligt.

- **Lieferverträge**: Langfristig vereinbarte Mindest- und Höchstabnahmenmengen und weitere, oft komplizierte Vertragsvereinbarungen
- **Leitungen**: Pipelinekapazitäten mit bestimmten Ober- und Untergrenzen
- **Gasqualitäten**: Gase mit unterschiedlichen Brennwerten (L- und H-Gas) können an Mischungsknoten gemischt werden, wobei bestimmte Zielbrennwerte einzuhalten sind.
- **Speicher**: In Poren- und Kavernenspeichern kann Gas unter Tage gespeichert werden und in Spitzenzeiten ins Netz eingespeist werden.
- **Diverse weitere Restriktionen**: Spotmärkte, gegenläufige Gasflussrichtungen, Kopplungen von Lieferanten und eine Reihe weiterer Vertragsgestaltungsoptionen

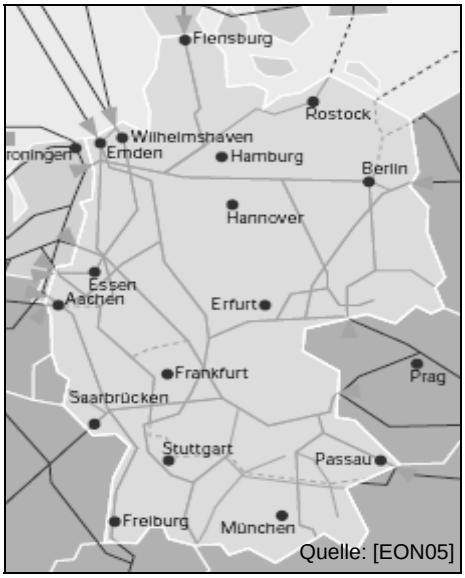

Abbildung 22: Das deutsche Ferngasnetz

Das beschriebene Minimierungsproblem wurde als MIP-Modell formuliert, in dem die Gasbezugskosten die Zielfunktion bildeten und die aufgezählten Nebenbedingungen als Restriktionen formuliert wurden. Hinzu kam eine zweistufige Modellierung, die aus einem Grob- und einem Feinmodell bestand, um bestimmte Nichtlinearitäten bei der Brennwertberechnung abzubilden. Details der Modellformulierungen seien hier aus Platzgründen ausgespart, ein zumindest vom Grundcharakter ähnliches Problem kann man aber bei [Tomlin88] nachlesen[1]. Vor EPOS wurde die Gaseinkaufsoptimierung mit einem etwas einfacheren Modell auf einem Großrechner mit dem Optimierungssystem MPSX und einem in PL/1 geschriebenen Modell-

[1] Tomlin beschreibt dort den Einsatz von Special Ordered Sets zur Modellierung z.B. von Bezugsfenstern bei Lieferanten oder zur Abbildung von Gasspeichercharakteristiken. EPOS enthielt in seinen ersten Versionen keine SOS. Deren Einsatz wurde allerdings für spätere Ausbaustufen positiv geprüft, etwa zur Modellierung von Druck-Mengenrelationen.

generator durchgeführt. Eine der wichtigsten Neuerungen des Entscheidungsunterstützenden Systems EPOS ist eine interaktive, grafische Benutzeroberfläche, die sowohl zur Visualisierung der Optimierungsergebnisse, als auch zur Editierung der Eingabedaten dient. Diese Oberfläche wurde größtenteils in Visual Basic 6.0 entwickelt, da sich diese Sprache sehr gut für das Rapid Application Development, besonders von GUIs, eignet (vgl. [McMahon00]). Die Ausgangsdaten des Modells werden aus dem Oracle-basierten Data-Warehouse des Unternehmens geladen und in einer applikationseigenen Datenbank („Storage") zwischengespeichert. Den Kern des Systems bildet die Optimierungsengine MOPS, die zunächst statisch über ein C-/Fortran-Interlanguage-Interface und später als DLL eingebunden wurde. Der Modellgenerator wurde unter Einsatz von Visual Studio 6.0 in C geschrieben.

Die Architektur des Systems ist durchaus typisch und kann – bezogen auf den Entwurfszeitpunkt – als „State-of-the-art" bezeichnet werden. Dessen ungeachtet und trotz der Tatsache, dass fast alle Projektbeteiligten jahrelange Erfahrung mit der Entwicklung von EUS oder Optimierungssoftware aufweisen konnten, traten bei der Erstellung des Modellgenerators, bei der Anpassung des Modells und der Integration der Optimierungssoftware diverse Problemfelder auf, die zusammen mit ihren Gründen im Folgenden aufgeführt werden sollen.

Zunächst kann eine relativ lange, sich über Monate hinziehende Entwicklungszeit des Modellgenerators angemerkt werden. Ebenso waren spätere Änderungen am Modellgenerator relativ zeitintensiv. Die Gründe hierfür lagen weniger am eingesetzten Solver selbst, als an fehlenden oder mangelhaften Umfeldfunktionen (Modellierungssprachen und -tools) sowie teilweise auch an falschen Softwaredesignentscheidungen.

Ein großer Teil der Probleme entstand durch fehlerhafte Ausgangsdaten oder durch Bugs in den Modulen, die die Daten aus dem Data-Warehouse in den lokalen Zwischenspeicher des EUS transferierten. Damit in Zusammenhang standen auch häufig Indizierungsfehler im Modellgenerator, weil z.B. auf eine Variable anstatt am Index j an j-1 oder j+1 zugegriffen wurde. Möglicherweise hätten sich einige dieser Fehler bei Einsatz von speziellen Datenstrukturen oder Klassen zur Modellierungsunterstützung vermeiden lassen. Solche Modellierungshilfsbibliotheken waren zum Projektzeitpunkt allerdings nicht verfügbar.

Ein anderes Problem waren die insbesondere zu Beginn der Modellgeneratorentwicklung häufig auftretenden Unlösbarkeiten, bedingt durch Programmier- oder Datenfehler. Zur besseren Analyse wurden künstliche, extrem hoch bepreiste Variablen eingefügt, die verhindern, dass das Modell unlösbar wird. Falls diese Variablen Werte größer Null annehmen, zeigt dies eine faktische Unlösbarkeit an. Dieses Verfahren war allerdings trotz Modellanalysetools des Optimierers unumgänglich. So enthält zwar MOPS beispielsweise Modelldebuggingfunktionali-

täten zum Auffinden des Irreducibly Infeasible Sets (IIS), diese sind aber bei MIP-Modellen problematisch.

Ein weiteres Problem war das zeitraubende Debugging des erzeugten Modells. Damit ist allerdings nicht die Analyse von Unlösbarkeiten gemeint, sondern einfach die Tatsache, dass es Umstände bereitete, die vom Modellgenerator erzeugte Matrix auf Korrektheit zu überprüfen. Hierzu wurde meist das Modell als Triplet ausgegeben und dann „manuell“ überprüft. Ausgabemöglichkeiten eines Debug-Modells in algebraischer Notation, etwa ähnlich dem LINDO-Format, oder noch besser das Vorhandensein eines Matrix-Viewers mit Sparse-Scrolling-Features hätte das Debugging des Modellgenerators erheblich beschleunigt.

Eine anderes zeitaufwändiges Prozedere war die Tatsache, dass Modellmodifikationen und Tests mit unterschiedlichen Modellvarianten immer eine Modifikation des C-Modellgenerators mit sich zogen, denn es existierte keine Möglichkeit, etwa in einer Modellierungssprache schnell unterschiedliche Modellvarianten zu implementieren und zu testen. Die hohe Performanz des Modellgenerators wurde somit durch einen Verlust an Flexibilität erkauft.

Zusammenzufassend kann man sagen, dass Modellgenerator und Optimierungssystem zwar durchaus ihre Funktion performant erfüllen und valide Ergebnisse liefern, dass aber der Prozess der Erstellung des Systems in diesem Kernbereich umständlich, fehlerbehaftet und damit suboptimal in Bezug auf die Entwicklungseffizient war. Die Gründe hierfür waren im Wesentlichen die Folgenden:

- **Kein Einsatz von Modellierungssprachen.** Vor allem für Tests und zum Ausprobieren unterschiedlicher Modellvarianten
- **Festhalten an prozeduraler Programmierung**. Während in andern Teilen des Systems objektorientierte bzw. -basierte Programmierung (C++/VB) erfolgreich zum Einsatz kam, wurde der Modellgenerator prozedural in C geschrieben. Grund hierfür war offenbar die Anlehnung an den in PL/1 geschriebenen Modellgenerator des Altsystems und die Tatsache, dass MOPS objektorientierte Konzepte bei der Modellgenerierung (noch) nicht unterstützt.
- **Fehlende Features bei Debugging-Tools**. Matrixgeneratoren sind prinzipiell schwer zu debuggen (vgl. [Fourer83]). Mit Debugging ist dabei weniger eine komplexe Unlösbarkeitsanalyse gemeint, als vielmehr die einfache Überprüfung der generierten Matrix. Hierzu kann u.a. ein spezieller Logging-Modus in MOPS aktiviert werden, und das Modell kann als MPS- oder Triplet-File ausgegeben und dann vom Entwickler überprüft werden. Es fehlt allerdings noch z.B. ein grafischer, interaktiver Modell-

explorer oder die Ausgabemöglichkeit spezieller Debug-Modelle. Letzteres ist bereits mit MOPS Studio möglich, Ersteres ist für die nächsten Versionen geplant.

- **Schwierige Verifizierbarkeit**. Das Risiko von Fehlern bei der Modellgenerierung ist für Matrixgeneratoren generell höher als bei der Verwendung von Modellierungssprachen (vgl. [Fourer83]). Besonders gravierend sind Fehler, die nicht zur Unlösbarkeit oder zu völlig unplausiblen Ergebnissen führen, sondern deren optimale Lösung zwar falsch, aber als solche nicht auf den ersten Blick erkennbar ist. Solche Verifizierungsprobleme gab es im Projekt EPOS zwar auch, allerdings in sehr geringem Umfang, da verlässliche Vergleichswerte aus dem Altsystem existierten und sehr viel Expertenwissen zur Lösungsüberprüfung vorhanden war.
- **Ex-Post-Dokumentation**. Es wurde zwar entsprechend den Regeln des Unternehmens eine Dokumentation für das Modell und den Generator erstellt, dies geschah jedoch nachträglich und separat (MS-Word). Es wurde kein integriertes Dokumentationssystem benutzt, und Wissen, z.B. über getestete Modellvarianten, existierte nur in den Köpfen der Entwickler oder in deren informellen Aufzeichnungen.
- **Tuning der Optimierungssystemparameter**. MOPS verfügt, wie andere Optimierer auch, über eine Vielzahl von Parametern zur Steuerung der Lösungsalgorithmen. Die Auswirkung eines Parameters auf die Performance und Lösungsqualität kann erheblich sein, ist aber meistens nur durch Ausprobieren festzustellen. Ein solches Parametertuning wurde erfolgreich durch den MOPS-Entwickler durchgeführt. Es wäre allerdings wünschenswert gewesen, wenn ein Tool zum automatisierten Tuning verfügbar wäre. Die EPOS-Entwickler haben sich hier mit einer eigenen kleinen Ad-hoc-Lösung auf VBA-Basis beholfen.
- **Fehlen von Hilfsbibliotheken**. Für bestimmte, in Modellgeneratoren häufig vorkommende Aufgaben (z.B. Stringsuche via Hashing oder dynamische, alphanumerisch indizierbare Arrays) wurden Routinen teils selbst erstellt. Man hätte zwar möglicherweise die eine oder andere Funktionalität auch aus anderen Standardbibliotheken benutzen können, was fehlte, war aber eine solvernahe spezielle Hilfsbibliothek mit derartigen Funktionen.

Die genannten Schwierigkeiten und ihre Gründe bilden den Hintergrund und Erfahrungshorizont eines realen Projekts für die folgenden, allgemeineren Aussagen.

5.2 Probleme der Integration von Optimierungssoftware in EUS

Der Fokus der Entwickler von Optimierungssystemen liegt in der Regel auf dem Numerischen Solver-Kern und weniger auf den Programmier- und Endbenutzerschnittstellen. Aus den in 2.2.3 genannten Gründen ist dies insbesondere bei akademischen und semikommerziellen Systemen der Fall. In diesem Abschnitt sollen zunächst allgemeine Schnittstellen- und Integrationsprobleme aufgezeigt werden, und anschließend wird auf systemspezifische Probleme von MOPS und anderen Optimierern eingegangen.

5.2.1 Allgemeine Probleme

Unabhängig von bestimmten Solvern oder Produkten können bei der Entwicklung von Entscheidungsunterstützenden Systemen die im Folgenden beschriebenen typischen Probleme in Bezug auf die Einbindung von Optimierungssoftware vorkommen.
Zunächst einmal seien die Schwierigkeiten genannt, die sich aus der Entwicklung eigener Modellgeneratoren gegenüber dem Einsatz einer Modellierungssprache ergeben und die teilweise schon in 2.2.4 aufgeführt wurden (vgl. auch [Fourer83]):

- Mangelhafte Verifizierbarkeit des erzeugten Modells
- Schwierige Modifizierbarkeit des Modells
- Aufwändige Dokumentation von Modell und Modellgenerator
- Abhängigkeit des Modellgenerators von den Datenstrukturen des Solvers
- Schwieriges Debugging
- Komplexität
- Keine scharfe Trennung von Modell und Daten

Diese allgemeinen Probleme von Modell- bzw. Matrixgeneratoren treten auf, wenn Modellierungssprachen nicht eingesetzt werden können, beispielsweise weil der verwendete Solver keine Schnittstellen zur gewünschten Modellierungssprache besitzt, aus Kostengründen, weil zur Steuerung des Lösungsprozesses volle Kontrolle über die Matrix nötig ist, oder weil die Transformation der Modelldaten in das Datenformat der Modellierungssprache vermieden werden soll.
Der Verzicht auf den Einsatz von Modellierungssprachen im Vorfeld des endgültigen Modells, beispielsweise, um unterschiedliche Modellformulierungen zu testen, bevor mit der Entwicklung des EUS begonnen wird, bringt erhebliche Effizienznachteile (vgl. [Kallrath04b,

S. 64]). Ebenso ist die langfristige Wartungsfreundlichkeit eines EUS bei Einsatz einer Modellierungssprache deutlich höher ([Kallrath04b, S. 65]).

Weitere potenzielle Probleme hängen mit den Eigenschaften der jeweils eingesetzten Solver und ihren Systembibliotheken zusammen und werden insbesondere bei der Modellgeneratorentwicklung relevant:

Hier ist zunächst das Fehlen objektorientierter Schnittstellen zu nennen. Allgemein anerkannt ist, dass objektorientierte Programmierung unter anderem zur Komplexitätsreduktion und zur Stabilitätserhöhung des Codes beiträgt (vgl. etwa [Booch94], [Meyer97]). Nicht alle Solver verfügen aber über objektorientierte Schnittstellen für die verbreiteten objektorientierten Sprachen (C++, Java, C#, VB.NET etc.). Eng damit in Zusammenhang steht auch das Problem einer mangelnden syntaktischen Intuitivität des Codes. Wie später (Kapitel 6) noch an Beispielen erläutert wird, erlauben objektorientierte Ausdrücke und Techniken, wie Operatorenüberladung, eine natürlichere und intuitivere Formulierung von Modellierungszusammenhängen, was die Lesbarkeit etwa eines Modellgenerators deutlich erhöht.

Entscheidungsunterstützende Systeme stehen mit anderen betrieblichen Informations- und Kommunikationssystemen in engen Datenaustauschbeziehungen. Fehlende Schnittstellen zu Datenbanken und zu Spreadsheetapplikationen können daher ein Problem darstellen, das sich allerdings in der Regel mit einem gewissen Programmieraufwand für Import- und Exportroutinen über Standardschnittstellen (z.B. ODBC) lösen läst. Die Nicht-Verwendung von spezifischen, optimierungssystemnahen Datenbank- und Spreadsheet-Interfaces stellt aber in der Regel Zusatzaufwand dar, verringert die Entwicklungseffizienz und erzeugt zusätzliche potenzielle Fehlerquellen. In [Fourer05] findet man Angaben über die Datenbank- und Spreadsheetschnittstellen aller wichtigen Optimierungssysteme.

Mit ganz wenigen Ausnahmen (vor allem *Open Solver Interface,* [COIN06]) sind die meisten Solverbibliotheken proprietär in Bezug auf eine spezifische Optimierungsengine und bieten keine Möglichkeit, andere Solver einzubinden. Problematisch ist dabei nicht nur die Einengung auf einen bestimmten Solver, sondern auch die Tatsache, dass die Entscheidung für einen Solver bereits vor Entwicklung des EUS, bzw. des Modellgenerators erfolgen muss. Ein Abwägen der Vor- und Nachteile mehrerer Solver unter Kosten- und Performancegesichtpunkten ist somit für das Modell nachträglich nicht möglich. Proprietäre Solverschnittstellen, bzw. fehlende Anbindungsmöglichkeiten anderer Solver stellen daher ein bedeutendes Problem für die EUS-Entwicklung dar.

Ein eher geringfügiges Problem dürfte die teilweise fehlende Plattformunabhängigkeit der Optimierungssysteme sein, denn einerseits sind viele Systeme bereits für mehrere Plattformen

erhältlich ([Fourer05]), und andererseits werden EUS häufig für nur ein Zielsystem entwickelt. Anders als bei Software, die für einen anonymen Massenmarkt produziert wird, ist bei EUS das relevante betriebliche Systemumfeld homogener und konstanter, so dass fehlende Plattformunabhängigkeit seltener ein gravierendes Problem darstellt.

Performance und Skalierbarkeit bei großen Modellen sind hingegen ein Problembereich, der erhebliche Auswirkungen auf die Leistungsfähigkeit des gesamten EUS haben kann. Gemeint ist dabei nicht die Performance des Optimierers, sondern die der Schnittstellensysteme inklusive der Modellgenerierungssprachen. Es ist ein großer Unterscheid, ob die generierte Matrix einige tausend Variablen und Restriktionen hat, oder ob es sich um Modelle mit einigen Millionen Objekten handelt. [Kallrath04b, S. 66] nennt Fälle, in denen die eigentliche Lösung eines MIP-Modells zwei bis drei Minuten dauerte, dessen Generierung dagegen 20 Minuten. Bei MOPS und MOPS Studio wurde Ähnliches beobachtet: Ein einfaches Matching-Modell mit 250.000 Variablen, 1.000 Restriktionen und 500.000 Nonzeros wird von MOPS innerhalb weniger Sekunden gelöst, die Generierung durch MathProg dauert jedoch ca. drei Minuten. Es liegt nahe, dass insbesondere Optimierungssysteme bzw. Modellierungssprachen aus dem nicht- oder semi-kommerziellen Bereich derartige Performance-Probleme bei sehr großen Modellen aufweisen.

Mängel in Zusammenhang mit der Indizierung, bzw. ein unkomfortables und/oder ineffizientes Indexmanagement, sei es in Modellierungsbibliotheken oder sei es in Modellierungssprachen, können vor allem bei sehr großen Modellen ebenfalls ein Problem darstellen. So können Praxisprobleme, z.B. aus dem Supply-Chain-Management-Bereich, oft mehr als zehn Indizes haben ([Kallrath04b, S. 66]). Vor allem in den Modellierungssprachen ist Indexing das wohl vorrangigste Thema (dazu beispielsweise [Kuip92], [Geoffrion92] oder [Hürlimann00]).

Die Modellgenerierungsmodule Entscheidungsunterstützender Systeme werden häufig zusammen mit der Modellformulierung in einem iterativen Prozess entwickelt, an dessen Anfang ein kleineres, komplexitätsreduziertes Modell steht, das bis zum vollständigen Endmodell ausgebaut wird. Während dieses Prozesses ist es wichtig, die Möglichkeit zu haben, ad hoc Modellierungsideen auszuprobieren. Ebenso müssen Teilprobleme manchmal losgelöst vom Hauptproblem modelliert, programmiert und gestestet werden. Programmierbare Spreadsheetanbindung, Anbindung an Modellierungssprachen, Funktionen zur Testdatengenerierung, Fehlertoleranz und automatische, intelligente Verwendung von Defaults (etwa bei Funktionsaufrufen) sind einige der Features, die diese Art der Ad-hoc-Modellierung und -Programmierung unterstützen und deren Fehlen zu Einbußen der Entwicklungseffizienz führt.

Schließlich sei noch auf das Problem des nicht durchgängigen Schnittstellengesamtkonzepts hingewiesen: Bei schon lange auf dem Markt befindlichen Optimierungssystemen sind neuere und fortschrittlichere Interfacekonzepte (wie z.B. ILOG's Concert Technology) einem aus Kompatibilitätsgründen nicht mehr änderbaren älteren Sockel von (meist prozeduralen) Schnittstellen hinzugefügt worden. Die einzelnen Interfacesysteme sind dabei untereinander syntaktisch und semantisch nicht mehr analog. Auch bieten oft nicht alle Schnittstellensysteme den vollen Funktionsumfang, was dazu führt, dass zuweilen für seltener benutzte Funktionen auf andere Schnittstellen zurückgegriffen werden muss (z.B. bei XPRESS). Ein mangelhaftes Schnittstellengesamtkonzept erfordert vom Programmierer, sich für ein und denselben Optimierer mehrere Syntax- und Semantik-Varianten einzuprägen und kann unsauberen Code mitverursachen.

5.2.2 Probleme spezifischer Optimierungssysteme

Neben den zuvor erwähnten allgemeinen Problemen besitzen die meisten der auf dem Markt befindlichen Solver noch spezifische Probleme bei ihrer Integration in EUS. Im Folgenden sollen einige Schwachstellen im Interfacebereich der Optimierer CPLEX, XPRESS und MOPS aufgeführt werden.

Eine der Schwierigkeiten beim Einsatz der ILOG Concert Technology ist beispielsweise ihre hohe Komplexität mit mehrfach verschachtelten Vererbungszusammenhängen und der Trennung von Handle- und Implementationsklassen. Da Mächtigkeit aber auch immer einen gewissen Grad an Komplexität bedingt, scheint das Komplexitätsproblem nicht gänzlich behebbar. Man muss sich aber fragen, inwieweit die Verhüllung von zwei unterschiedlichen Optimierungsengines (CPLEX und SOLVER) unter einer gemeinsamen Schnittstellenschicht unter Inkaufnahme einer hohen Komplexität tatsächlich erforderlich ist. Weiterhin kann aus technischer Sicht die enge Verzahnung der ILOG Produkte CPLEX, SOLVER, Concert, OPL und OPL Studio kritisieren werden, die nicht über offene Schnittstellen miteinander verbunden sind und die die Interaktion mit Fremdprodukten, etwa die Anbindung fremder Solver an OPL schwer bis unmöglich machen. Aus Gründen der Marktstrategie und der Kundenbindung ist diese Architektur allerdings nachvollziehbar.

Bei der XPRESS Builder Component Library kann die Vermischung von objektorientierter und prozeduraler Programmierung negativ gesehen werden: Bei Benutzung der BCL, etwa innerhalb von objektorientiertem C++-Code, kann es vorkommen, dass prozedurale Funktionen der Callable Library des Solvers XPRESS benutzt werden müssen, da nicht alle, insbesondere nicht die solvernahen Funktionalitäten in der BCL abgebildet sind. Dies ist zwar si-

cherlich kein gravierendes Problem, zeigt aber einen Mangel an Einheitlichkeit des Schnittstellenkonzepts an.

Auch der Optimierer MOPS hat im Bereich der Schnittstellen und der Usability noch einige Schwachstellen, auch wenn er in den vergangenen Releases große Fortschritte gemacht hat, was den numerischen Kern und die Lösungsperformance anbelangt und hier in vielen Fällen gleichauf mit den kommerziellen Marktführern ist (vgl. [Suhl05], [Koberstein05]).

Beispielsweise besitzen das statische Interface und die MOPS.DLL zwar semantisch sehr ähnliche Funktionssets, sind aber syntaktisch nicht analog. Das betrifft vor allem die Funktionsnamen: Während die DLL-Funktionen über intuitive Namen wie z.B. *WriteMpsFile* angesprochen werden, nutzt das IMR-Interface MOPS-interne, kryptische Bezeichnungen (z.B. *xstmps*) für ansonsten semantisch gleiche Funktion. Dies ist ein Fall des schon zuvor angesprochenen Problems der fehlenden einheitlichen, durchgängigen Schnittstellenkonzeption.

Ähnlich wie die Namen der IMR-Funktionen sind auch die Parameterbezeichnungen benutzerunfreundliche MOPS-interne Kürzel, die über eine nur Strings verarbeitende Routine (*SetParameter*) gesetzt werden, was z.B. bei Fließkommawerten zu Konvertierungsverlusten führen kann. Dieses Problem wurde allerdings ab Version 9.x durch zusätzliche Parameterverwaltungsfunktionen behoben.

Ein weiterer Mangel ist die fehlende Möglichkeit zur parallelen Verwaltung und Optimierung mehrerer Modelle, wie es in Szenariorechnungen oder etwa bei Column Generation-Problemen erforderlich ist. Dies ist nicht nur ein Problem des numerischen Kerns, sondern auch der diese Modelle verwaltenden Schnittstellenschicht. Syntaktisch hat dies bei MOPS den Verzicht auf Environments und Modelpointer zur Folge. Gäbe es die Möglichkeit der parallelen Benutzung mehrerer Modelle, so wäre auch Multithreading-Unterstützung wünschenswert, denn dann könnten parallele Modellgenerierungen und Optimierungen ablaufen, was insbesondere auf Mehrkernprozessoren positive Skalierungseffekte hätte.

MOPS besitzt nur prozedurale Interfaces. Diese lassen sich zwar, dank der universellen Einbindbarkeit der DLL, auch aus objektorientierten Sprachen, wie C++, C#, Java oder VB.NET, ansprechen, es fehlen aber spezifische objektorientierte Konzepte, wie sie beispielsweise die Concert Technology oder XPRESS BCL bieten. Gleiches gilt auch für die fehlende Anbindung an Modellierungssprachen, die erst im Rahmen dieser Arbeit geschaffen wurden. Ebenso wurden in diesem Zusammenhang Callbacks eingeführt, die in MOPS-Versionen vor 9.x fehlten und die für vielfältige Statusabfragen und einen benutzergesteuerten Abbruch des Optimierungslaufs genutzt werden können.

Die Reporting-, Debugging- und Import-/Export-Möglichkeiten für Modelle und Daten sind begrenzt. Es fehlen Hilfsbibliotheken zum Datenmanagement, Import- und Exportmöglichkeiten für diverse Modellformate (z.B. MPS-Modelle mit ILOG's Formaterweiterungen, Lindo etc.), komfortable Debugging-Ausgabe von Modellteilen und Reportgeneratorunterstützung für Lösungs- und Datenausgabe.
Spreadsheetbasierte Ad-hoc-Modellierung ist durch das Excel Plug-In *ClipMOPS* möglich, allerdings fehlt dessen Ansteuerungsmöglichkeit aus VBA-Script. Ebenso existierte vor MOPS Studio kein integriertes Modellierungssystem, in dem ein schnelles Ad-hoc-Testen von Modellformulierungen möglich ist.
Schließlich sei noch auf einige allgemeine softwaretechnische Probleme hingewiesen, die alle Solver betreffen, die als DLL unter Windows eingebunden werden (u. a. CPLEX, XPRESS, MOPS). Diese Probleme sind allgemeine Schwierigkeiten der Verwendung von DLLs (ausführlich dazu [Brockschm96]):

- Versionierungsprobleme (*„DLL Hell"*, siehe auch [Szyperski99])
- Pfad- und Namensabhängigkeiten
- Keine Gliederung der Funktionen in Interfaces
- Zwang zur Verwendung von Funktionsdeklarationen auch in Sprachen, die wie Visual Basic normalerweise keine expliziten Header benötigen
- Zwang für Client und Server/Solver zum Ablauf im gemeinsamen, evtl. begrenzten Prozessraum, keine Möglichkeit der Verteilung auf getrennte Prozesse oder auf unterschiedliche Rechner wie etwa bei COM

5.3 Anforderungen an Optimierungssoftware-Schnittstellen

Die Anforderungen an Optimierungssoftware und ihre Schnittstellen ergeben sich zum einen aus allgemeinen Softwarequalitätsanforderungen und zum anderen aus spezifischen Erfordernissen, die für Optimierungssysteme gelten, Letzteres vor allem vor dem Hintergrund der Einbettung in EUS. Die spezifischen Anforderungen sind dabei unmittelbare Schlussfolgerung der im vorangegangen Unterkapitel aufgeführten Probleme der Integration von Optimierungssoftware in EUS und werden ergänzt um usability-bezogene Anforderungen für integrierte Optimierungssysteme.

5.3.1 Allgemeine Anforderungen an Software

Die allgemeinen Qualitätsanforderungen an Software sind in verschiedenen internationalen Normen festgehalten. Zunächst existieren die sehr unspezifischen Qualitätsmanagementnormen DIN-ISO 9000-9004, die aber nicht in ihrer Gesamtheit auf die Softwareentwicklung anwendbar sind und die zudem auf Softwareproduktionsprozesse und nicht auf Produkte abzielen (zu den für die Softwareentwicklung relevanten Teilen der ISO 9000-9004 siehe [Bächle96]).

Die Beurteilung der Qualität eines fertigen Softwareprodukts erfolgt vielmehr nach ISO/IEC 9126 bzw. DIN 66272. Diese Normen formulieren ein Begriffssystem zur Beurteilung von Softwarequalität und definieren somit Anforderungskriterien, denen eine normgerechte und damit qualitätssuffiziente Software entsprechen muss. Dazu werden allerdings keine konkreten Maße, Messmethoden oder Einstufungen vorgegeben, da diese je nach Softwaretypus unterschiedlich sind. Die Software-Qualitätsmerkmale und Untermerkmale der ISO/IEC 9126 sind (nach [Hohler94] zitiert aus [Krcmar00, S. 125]):

- **Funktionalität**
  - Angemessenheit
  - Richtigkeit
  - Interoperabilität
  - Ordnungsmäßigkeit
  - Sicherheit
- **Zuverlässigkeit**
  - Reife
  - Fehlertoleranz
  - Wiederherstellbarkeit
- **Benutzbarkeit**
  - Verständlichkeit
  - Erlernbarkeit
  - Bedienbarkeit
- **Effizienz**
  - Zeitverhalten
  - Verbrauchsverhalten
- **Änderbarkeit**
  - Analysierbarkeit
  - Modifizierbarkeit

  - Stabilität
  - Prüfbarkeit
- **Übertragbarkeit**
  - Anpassbarkeit
  - Installierbarkeit
  - Konformität
  - Austauschbarkeit

Ähnliche Kriterien findet man auch in der Literatur; beispielsweise definiert [Meyer97, Kap. 1] folgende Softwarequalitätsanforderungen: Correctness, robustness, extendibility, reusability, compatibility, efficiency, portability, ease of use, functionality, timeliness.
Diese Kriterien bilden einen sehr allgemeinen Anforderungsrahmen, dem alle Arten von Optimierungssoftware entsprechen sollten. Die nachfolgenden beiden Unterkapitel konkretisieren einige dieser Anforderungen für den Bereich der Programmier- und Endbenutzerschnittstellen.

5.3.2 Anforderungen an Programmierschnittstellen

Die hier aufgeführten Anforderungen betreffen die von Optimierungsengines sowie ferner die von Modellierungssprachen und sonstigen einbettbaren Modellierungshilfsbibliotheken angebotenen Programmierschnittstellen.
Die wohl wichtigste Anforderung ist die **Zurverfügungstellung eines einheitlichen Schnittstellensystems mit mehreren zielkontext-adäquaten Ausprägungen**. Den Zielkontext bildet dabei die zur Implementation verwendete Programmiersprache, evtl. ein Framework (z.B. .NET) und ggf. eine Applikationsumgebung (z.B. Excel, Matlab etc.). Mit zielkontext-adäquater Ausprägung ist gemeint, dass die einzubettende Optimierungssoftware eine Ausprägung ihres Schnittstellensystems besitzt, die sich bezüglich Programmierparadigma, Syntax, Framworkabhängigkeit etc. möglichst konform, intuitiv und „natürlich“ aus dem Zielkontext benutzen lässt und ggf. spezifische Features des Zielkontextes ausnutzt. Zielkontext-adäquat wäre beispielsweise ein Optimierungssystem, das für prozedurale Programmiersprachen eine prozedurale Version seines Schnittstellensystems anbietet und für objektorientierte Sprachen eine objektorientierte Version, wobei Letztere zwar spezifische Vorteile der Objektorientierung ausnutzt, aber dennoch größtmögliche Analogie zu allen andern Schnittstellen des Systems bewahrt.

Die Anforderung der **Plattformunabhängigkeit** impliziert ein Plattform übergreifendes einheitliches Schnittstellensystem, das aber aufgrund der sehr unterschiedlichen Programmierkontexte nur auf einem kleinsten gemeinsamen Nenner umsetzbar sein dürfte. Bei den meisten derzeitigen Systemen wird Plattformunabhängigkeit lediglich über gleiche oder ähnliche Kommandozeilenschnittstellen erreicht, weniger jedoch im Bereich der eigentlichen Programmierschnittstellen.

Aus der schon mehrfach erwähnten Vorteilhaftigkeit der **Modellierungssprachenverwendung** erwächst die Anforderung der Modellierungssprachenanbindung aus dem Programmcode, wobei Zugriff und Veränderung der Bestandteile (Variablen, Restriktionen Indizes, etc.) eines durch Modellierungssprachencode generierten Modells auch aus Programmcode heraus möglich sein sollten. Im Zusammenspiel von Solver und Modellierungssprache sollte der Datenaustausch in beide Richtungen einfach sein (vgl. [Schichl04, S. 51]). Eine Kombination unterschiedlicher Solver und Modellierungssprachen über eine einheitliche Programmierschnittstelle wäre wünschenswert. Ebenso die Verfügbarkeit von Hilfsfunktionen zur Generierung des für die Modellierungssprachen erforderlichen Dateninputs (z.B. Textfiles).

Auch wenn marktstrategische Gründe (Kundenbindung) dagegen sprechen mögen, sollten **mehrere Solverengines** und mehrere Modellierungssprachen über ein einheitliches Schnittstellensystem ansprechbar sein, was die oben genannte ISO-Anforderung „Übertragbarkeit“ konkretisieren würde. Faktisch dürfte diese Anforderung allerdings schwer umsetzbar sein, da nicht alle Hersteller ihre Low-Level-Interfaces freilegen und eine Einigung auf ein gemeinsames Interfacesystem unter Konkurrenten eher unwahrscheinlich ist. Ist kein gemeinsames, einheitliches Schnittstellensystem erreichbar, sollten nach Möglichkeit „Kompatibilitäts-Interfaces“ die Austauschbarkeit der Solver oder Modellierungssprachen erleichtern. Dies könnte realisiert werden, indem ein Optimierungssystem zumindest ein Subset der Schnittstellen eines anderen Systems emuliert.

Hilfsbibliotheken sollten einen einfachen und fehlertoleranten Datenaustausch mit Datenbanken, Spreadsheets und anderen Datenbehältern wie etwa CSV- oder XML-Dateien ermöglichen. Zwar enthalten die meisten modernen Programmierumgebungen bereits Unterstützung für solche Zugriffe, die entsprechenden Funktionen, Klassen oder Komponenten sind aber allgemeiner Natur. Optimierungssystembibliotheken sollten diese allgemeinen Datenzugriffsmechanismen erweitern oder eigene Funktionalitäten mitbringen, so dass der Datenfluss zwischen Optimierungssystem und Datenquelle mit möglichst wenig zusätzlichem Programmieraufwand möglich ist.

Als Folgerung der bereits erwähnten Gefahr von Performance- und Skalierungsproblemen, vor allem bei nicht kommerziellen Systemen, gilt als weitere wichtige Anforderung, dass Schnittstellen-, Modellverwaltungs- und Modellgenerierungssysteme effizient und **performant implementiert** sind. Damit in Zusammenhang steht auch die Forderung nach einem stabilen Indexing, das vielfache Indizierung und sehr große Indexmengen effizient bewältigt und einfach aus der jeweiligen Programmiersprache heraus nutzbar ist. Zur Erleichterung des Arbeitens mit Indexmengen sollte es ggf. zusätzliche Hilfsklassen o. ä. geben.
Ähnlich wie beim Rapid Application Development, sollte ein Optimierungssystem in seinen Programmierschnittstellen ein „**Rapid Model Development**“ ermöglichen. Bei dieser Art der Ad-hoc-Modellgenerierung sollte die programmatische Modellerstellung mit möglichst wenigen Befehlen erfolgen. Hilfreich hierzu wären beispielsweise Konstrukte, die Schleifen ersetzen, Indexmengen handhaben, einfachen Direktzugriff auf Datenbestände ermöglichen und Datentransformationen und Berechnungen für Arrays möglichst iterationsfrei mit einem Befehl erlauben.
Schließlich seien noch einige **weitere Anforderungen** erwähnt, die zwar von den meisten kommerziellen, aber keineswegs von allen Optimierungssystemen erfüllt werden:

- Möglichkeit der gleichzeitigen Erstellung und Optimierung mehrerer Modelle
- Multithreading-Fähigkeit
- Lauffähigkeit in 64-Bit-Umgebungen
- Callbacks für Statusinformationen und Lösungsprozesssteuerung
- Funktionen zum Lesen und Schreiben möglichst vielfältiger Modellformate
- Debugging-Unterstützung (z.B. Logging, Exceptions etc.)

5.3.3 Anforderungen an Endbenutzerschnittstellen

Die wichtigste Anforderung an ein integriertes Modellierungssystems ist es, dem Enduser eine effiziente und bequeme grafische Benutzeroberfläche zu bieten, die den Modellierungszyklus (siehe 3.3) möglichst weitgehend abdeckt und Modelle, Solver und Utilities verbindet.
In Anlehnung an [Schichl04, S. 60] können folgende Anforderungen für die GUIs integrierter Modellierungssysteme formuliert werden:

- Grafische mathematische Notation wie etwa im Programm *Mathematica* (mit Symbolen wie z.B. $\sum, \leq, \geq$)
- Ansichten der Modellstruktur mit unterschiedlicher Granularität (fein, grob)
- Syntax-orientiertes Editing (Syntax Coloring, automatisches Vervollständigung, Syntaxcheck in Echtzeit etc.)

- Interaktive Hilfe
- Visualisierung der Zusammenhänge zwischen Modellteilen mit Kreuzreferenzen
- Visualisierung des Lösungsprozesses
- Visualisierung der Ergebnisse

[Schichl04, S. 62] erwähnt ferner, dass diverse große Modelle aus den Bereichen Chemie, Fertigung und Scheduling aus mehreren so genannten *Building Blocks* bestehen. Building Blocks sind kleinere, weitgehend autonome Modellteile, die über einzelne Restriktionen verbunden werden. Endbenutzerschnittstellen integrierter Modellierungssysteme sollten Unterstützung für das Arbeiten mit solchen Building Blocks bieten.
Weiterhin betont [Schichl04, S. 62] die Forderung nach Unterstützung für ein offenes Modellaustauschformat, das – anders als MPS – keine Rundungsprobleme bei der Umwandlung von Fließkommazahlen in Strings und umgekehrt hat. Ein solches Format könnte XML- oder MathML-basiert sein und sollte sowohl eine ASCII-, als auch eine binäre Variante haben und zum Schutz geistigen Eigentums verschlüsselbar sein.
Weitere Anforderungen an integrierte Modellierungssysteme aus Endnutzersicht resultieren aus vorteilhaften Features bestehender Systeme und der Literatur:

- Projekte zur Zusammenfassung von Modellen, Daten und Einstellungen
- Modelexplorer
- Dataexplorer und Anbindung an Datenbanken, Spreadsheets und Dateiformate mit grafischer Unterstützung (z.B. Tabellenansichten, Drag-and-Drop etc.)
- Page templates (ähnlich AIMMS, vgl. [Bisschop04, S. 76 f.])
- Case- und Szenariomanager für Modellvarianten
- Anbindung verschiedener Solver und ggf. verschiedener Modellierungssprachen
- Unterstützung für verteile Optimierung (ggf. agentenbasiert wie in AIMMS, vgl. [Bisschop04, S. 98 ff.])
- Intelligente Fehlermeldungen mit Korrekturvorschlägen
- Debugging (z.B. Stepping durch prozedurale Parts von Modellierungssprachencode)
- Grafischer Matrix-Explorer mit Zoom- und Scroll-Funktionen (wie etwa in MPL, vgl. [Kristjansson04, S. 249])
- Erweiterte Visualisierungsmöglichkeiten für Ergebnisse (wie z.B. Gantt Charts zur Visualisierung von Scheduling-Problemen oder „Christmas Trees“ in OPL Studio, die allerdings teilweise nur für CP-Modelle sinnvoll sind, vgl. [vHentenr04])