3 State-of-the-art von Solver- und Modellierungssystemschnittstellen

Dieses Kapitel beschreibt den aktuellen Stand der Technik in Bezug auf Schnittstellen zu Solvern, Modellierungssystemen und Modellierungssprachen. Dabei wird über eine rein deskriptive Darstellung der untersuchten Systeme hinausgegangen, indem strukturierende Einordnungen anhand von Funktionalitäten und anderen Eigenschaften vorgenommen werden. Außerdem wird versucht, ausgehend von diesen Einordnungen, gemeinsame Merkmale herauszukristallisieren, die als generische Features den produktunabhängigen Stand der Technik repräsentieren.

Was den Begriff „State-of-the-art“ anbelangt, so wird dieser hier im Sinne von „aktueller, verbreiteter Stand der Technik“ benutzt, und nicht in dem allumfassenden Sinn, wie ihn beispielsweise das Europäische Patentamt anwendet ([EPO06]). Dort wird unter dem State-of-the-art alles verstanden, was in irgendeiner Weise veröffentlicht oder in Gebrauch ist, also auch jede noch so kleine Innovation oder bloße Besonderheit eines Produkts. Im Folgenden soll hingegen vielmehr ein „Common State-of-the-art“ beschrieben werden, also der Stand der Technik, wie ihn die führenden Systeme *gemeinsam* darstellen.

Entsprechend der Zielsetzung, die Einbettung von Optimierungssystemen in EUS zu untersuchen, liegt der Fokus der Analyse dabei nicht auf den internen algorithmischen Fähigkeiten der unterschiedlichen Optimierungssysteme, sondern auf der Art und Weise ihrer Einbettungsmöglichkeiten. Es werden daher ausschließlich die Benutzer- und Programmierschnittstellen untersucht und nicht, ob und wie bestimmte Optimierungsalgorithmen implementiert sind oder wie beispielsweise das Performanceverhalten der Solver ist.

Einen Überblick über die aktuellen Solver- und Modellierungssysteme geben [Fourer05] und [Kessler04], sowie in sehr knapper Form [Klein04]. Aus der Marktübersicht in [Fourer05] wird erkennbar, dass die meisten (31 von 43) der aufgelisteten Optimierungssysteme als eigenständige Applikation ausführbar sind. Eine fast ebenso große Zahl, nämlich 28, ist als prozedural aufrufbare Bibliothek (Callable Library) verfügbar, während nur 18 über Klassenbibliotheken bzw. objektorientierte Schnittstellen verfügen. Darüber hinaus besitzt rund die Hälfte auch Add-In-Schnittstellen, insbesondere zu Excel und Matlab. Die bedeutenden kommerziellen Systeme bieten dabei in der Regel mehrere Schnittstellentypen, beispielsweise sowohl prozedurale, als auch objektorientierte Bibliotheken, sowie teilweise sehr leistungsfähige, eigenständige, integrierte Modellentwicklungs- und Modellverwaltungssysteme, die im Kern den jeweiligen Solver benutzen. Zu den Modellierungssystemen zählen auch die algebrai-

schen Modellierungssprachen, die teils solverunabhängig (z.B. AMPL oder GAMS), teils aber auch an bestimmte Solver gekoppelt sind (z.B. OPL).
Die folgende Beschreibung und Analyse des Stands der Technik im Bereich Solver- und Modellierungssysteme gliedern sich nach der Schnittstellentypisierung in die vier Bereiche

- Prozedurale Optimierungssystemschnittstellen
- Objektorientierte Optimierungssystemschnittstellen
- Modellierungssprachen
- Integrierte Optimierungssysteme

Für jeden dieser Bereiche werden einige bekannte und verbreitete Repräsentanten ausgewählt. Die beiden Marktführer für lineare Optimierung *ILOG CPLEX* und *XPRESS-MP* eignen sich für eine derartige Untersuchung besonders, da beide Produkte sowohl über prozedurale und objektorientierte Bibliotheken verfügen, als auch aus den integrierten Modellierungssystemen *OPL Studio* bzw. *XPRESS IVE* mit den Modellierungssprachen *OPL* bzw. *MOSEL* benutzbar sind.
Die ausgewählten Optimierungssysteme werden mit Bezug auf ihre Interaktions- und Einbettungsmöglichkeiten in Entscheidungsunterstützende Systeme kategorisierend gegenübergestellt. Ergebnis der Analyse ist eine Beschreibung des Common State-of-the-art anhand gemeinsamer generischer Features bezogen auf das Schnittstellen- und Interaktionsdesign.

3.1 Prozedurale Optimierungssystemschnittstellen

3.1.1 Untersuchungsrahmen

Prozedurale Schnittstellen sind funktionsbezogen, d.h. Aufrufe von Funktionen oder Prozeduren, und damit Aktionen oder Prozesse, stehen im Vordergrund, nicht der Umgang mit Objekten und ihren Methoden und Eigenschaften. Die Einbindung linearer Optimierer in andere Programme erfolgt sehr häufig über prozedurale Bibliotheken, weil dies der universellste Weg ist, Optimierungsfunktionalität für fast alle Programmiersprachen – auch für objektorientierte – bereitzustellen. Unter Windows werden die Funktionen und Prozeduren einer Callable Library meist als dynamische Bibliothek (DLL) eingebunden, seltener als statische LIB, wobei manche Solver über beide Bibliothekstypen verfügen (z.B. MOPS). Für weitere Details sei auf Kapitel 4 verwiesen, wo ausführlich aus softwarearchitektonischer Sicht auf die Unterschiede zwischen objektorientierter und prozeduraler Programmierung eingegangen wird.

Zur Darstellung des Stands der Technik im Bereich der prozeduralen Schnittstellen wurden die Callable Libraries der beiden Marktführer CPLEX und XPRESS-MP sowie das ebenfalls weit verbreitete Optimierungssystem LINDO ausgewählt. Diese Auswahl erfolgte aufgrund der Verbreitung und des Bekanntheitsgrades besagter Systeme und aufgrund des Umfangs und der Mächtigkeit der genannten Callable Libraries. CPLEX, XPRESS und LINDO als State-of-the-art-Systeme wurden beispielsweise auch von Atamtürk und Savelsbergh für ihre vergleichende Analyse von Optimierungsparametern in [AtaSav05] heranzogen.
Die Vorgehensweise zur Untersuchung der prozeduralen Bibliotheken besagter Solversysteme war Folgende: Zunächst wurden die entsprechenden Dokumentationen ([Dash02], [ILOG03], [LINDO06]) nach Funktionen durchsucht, die die lineare Optimierung betreffen und aufgelistet. Die genannten Optimierer bieten darüber hinaus auch eine Vielzahl von Funktionen für nicht-lineare Optimierung, die hier jedoch ausgeblendet bleiben. Anschließend wurden die aufgelisteten Bibliotheksfunktionen den folgenden Funktionskategorien (angelehnt an [ILOG03b, S. 114]) zugeordnet:

1. Umgebungs- und Modellhandling
2. File I/O
3. Modellaufbau und -modifikation
4. Modellabfrage
5. Lösungsabfrage und -analyse
6. Lösungsprozesssteuerung
7. Parameterhandling
8. Callbacks
9. Fehlermeldungen
10. Sonstige Funktionen

Innerhalb dieser Kategorien wurden dann einander entsprechende Funktionen der unterschiedlichen Solver gesucht, um herauszufinden, welche Funktionen produktspezifisch sind und welche eher universellen Charakter für Solverbibliotheken haben.
Ergebnis der Auswertung (s. Anhang 1) ist, dass die prozeduralen Bibliotheken der untersuchten Systeme eine große Menge an gleichen oder zumindest sehr ähnlichen Funktionalitäten besitzen, die in Abbildung 7 als Schnittmengen skizziert sind. Die gestrichelt eingezeichnete Menge symbolisiert die Menge der generischen Solverfunktionen, die einen *Common State-of-the-art* darstellen. Damit ist der Funktionsumfang gemeint, den moderne Solverbibliotheken derzeit typischerweise besitzen. Bei dieser Menge handelt es sich grob um die Vereini-

gungsmenge der Schnittmengen, also alle Funktionen, die in mindestens zwei Bibliotheken vorhanden sind, nicht jedoch um rein produktspezifische Funktionen. Allerdings ist eine trennscharfe Abgrenzung der Funktionen des *Common State-of-the-art* nicht immer möglich, da manche Konzepte (z.B. Callbacks, Cuts etc.) zwar prinzipiell von mehreren Systemen unterstützt werden, aber in den untersuchten Solverbibliotheken sehr unterschiedlich implementiert sind und eine 1:1 Zuordnung der Funktionen daher scheitert.

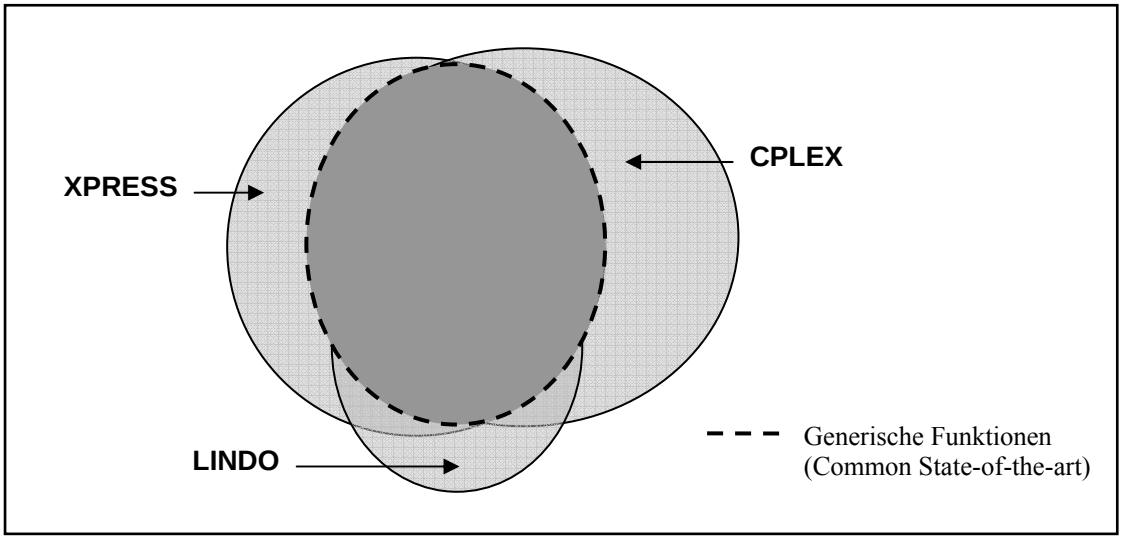

Abbildung 7: Schematisierung Funktionsüberschneidungen Callable Libraries

Die genannten Bibliotheken weisen einen sehr unterschiedlichen Funktionsumfang auf: Die mit Abstand umfangreichste Bibliothek ist die CPLEX Callable Library mit 280 Funktionen für LP- und MIP-Optimierung, es folgen die XPRESS-Bibliothek mit 131 Funktionen und LINDO mit 113 Funktionen. Es ist aber keineswegs so, dass von einer größeren Funktionszahl automatisch auch auf eine entsprechend größere Gesamtleistungsfähigkeit einer Optimierungsbibliothek geschlossen werden kann. Ein Grund hierfür ist unter anderem der unterschiedliche Umgang mit Parametern:

Alle untersuchten Solver benutzen eine Vielzahl von Parametern, manchmal auch „Attribute“ genannt, die die verschiedenen Lösungsalgorithmen steuern oder diverse Informationen über Lösung und Modell liefern. Parameter können entweder über universelle Parameterfunktionen der Art

```
GetParameter(ParameterID, Value)
SetParameter(ParameterID, Value)
```

behandelt werden, oder es gibt für jeden einzelnen Parameter gesonderte Funktionen

```
GetParameterX(Value)
SetParameterX(Value)
```

Während beispielsweise CPLEX zum Auslesen der Modelldimensionen (Zeilen, Spalten, Nonzeros) drei gesonderte Funktionen bereitstellt, verfügt XPRESS über keine explizite Funktion, sondern liest die Dimensionen über die universelle `XPRSgetintcontrol`-Routine aus. Dennoch ist XPRESS in diesem Beispiel aufgrund der fehlenden Funktionen nicht weniger leistungsfähig als CPLEX, allenfalls ist der Programmierkomfort bei Vorhandensein expliziter Funktionen geringfügig höher. Zwischen Anzahl der Funktionen einer Optimierungsbibliothek und deren Gesamtleistungsfähigkeit herrscht also keine strenge Korrelation. Die Anzahl der offerierten Funktionen ist lediglich ein grober Indikator für die Gesamtleistungsfähigkeit einer Bibliothek.

Da die Anzahl der Parameter und der mit ihnen verbundenen Funktionalitäten sehr groß ist, wurden in die Auswertung in Anhang 1 nur solche Funktionalitäten einbezogen, für die in mindestens einer Bibliothek explizite Funktionen vorhanden sind. Häufig betreffen die Parameter aber ohnehin nur die eher solver-spezifische Feinsteuerung von Optimierungsalgorithmen und sind daher für die Darstellung gemeinsamer generischer Features von prozeduralen Solverbibliotheken wenig bedeutsam.

3.1.2 Funktionalitäten

3.1.2.1 Umgebungs- und Modellhandling

Bevor mit der eigentlichen Modellgenerierung begonnen werden kann, erfordern alle Solver den Aufruf bestimmter Initialisierungsprozeduren, in denen interne Variablen und Speicherbereiche initialisiert werden und das Vorhandensein von Lizenzfiles überprüft wird.

Je nachdem, ob Modelle direkt oder innerhalb eines so genannten *Environments* erzeugt werden, lassen sich zwei Gruppen von Solvern unterscheiden: CPLEX und LINDO arbeiten mit einem oder mehreren Environments, die jeweils wiederum eines oder mehrere Modelle enthalten können. XPRESS-MP hingegen kennt keine Environments und benötigt nur den Aufruf von solver- aber nicht environment-bezogenen Initialisierungsroutinen, bevor ein Modell erzeugt werden kann. Das Konstrukt des Environments bedeutet eine zusätzliche Ebene, die vorteilhaft sein kann, wenn mehrere Modelle gleichzeitig im Speicher gehalten und organisiert werden müssen, denn es können z.B. globale Environment-Parameter gesetzt werden, die dann für alle Modelle dieses Environments gültig sind.

Da die Environments mithilfe von Pointern abgebildet und unterschieden werden, ist zunächst immer der Aufruf einer Funktion notwendig, um eine Umgebung einzurichten und einen entsprechenden Environment-Pointer zu erhalten (z.B. `LScreateEnv` in LINDO). Das ohne Environments arbeitende XPRESS-MP erfordert den Aufruf der Initialisierungsfunktion

`XPRSinit`, die vor allem der Überprüfung der Lizenzfiles dient, aber keinen Environment-Pointer liefert. Nach Erzeugung einer Umgebung wird das eigentliche Modell angelegt, wobei die entsprechenden Funktionen (z.B. mit `CPXcreateprob` in CPLEX) neben einem Status-Returncode einen Pointer auf das neu angelegte Modell liefern.
Die Möglichkeit mit mehreren Environments und Modellen gleichzeitig zu arbeiten, erfordert, dass bei allen weiteren Funktionsaufrufen das Environment und das Modell jeweils über ein Pointerargument spezifiziert wird, was zwar nicht vermeidbar ist, aber die Funktionsaufrufe der Callable Libraries aufbläht.
Die Möglichkeit des parallelen Handlings mehrerer Modelle bietet unbestreitbar Vorteile, beispielsweise in Applikationen, in denen Szenarien oder zusammengehörige Modellgruppen behandelt werden müssen. Auch die Lösung von Column-Generation-Problemen erfordert in der Regel, dass mindestens zwei Modelle gleichzeitig gehalten werden, da ein Haupt- und ein Untermodell zur Bestimmung der neu in das Hauptmodell einzufügenden Spalten, ausgehend von dessen Dualwerten, alternierend gelöst werden (vgl. [Desr05] als Einführung sowie [Gilmore61] und [Gilmore63] zu einem typischen Problem).
Zur Unterstützung des Handlings mehrerer Modelle bieten CPLEX und in geringerem Ausmaße XPRESS diverse Funktionen zum Kopieren von Informationen von einem Modell in ein anderes. Hierbei können komplette Modelle, wie auch bestimmte einzelne Modellinformationen (Parameter, Namen, Knotenprioritäten, Startbasis etc.) kopiert werden. Derartige Kopierfunktionen sind insbesondere dann hilfreich, wenn mehrere, einander ähnliche Modelle gelöst werden sollen, wie etwa bei Szenariorechnungen, in denen sich nicht die grundsätzliche Modellstruktur, sondern nur einige Ausgangsdaten ändern.

3.1.2.2 File I/O

Ein- und Ausgabe von Modellen und modellbezogenen Informationen in Dateien sind zentrale Funktionalitäten, über die alle Solverbibliotheken verfügen.
Ursprünglich war das Einlesen von Modellen aus Lochkarten der Haupteingabeweg[1]. Aus diesem historischen Grund hat sich das stark an Lochkarten orientierte MPS-Format (vgl. [MPS05]) für fast alle Optimierter als Standardformat durchgesetzt und wird in fast allen Optimierungsbibliotheken mit Schreib- und Lesefunktionen unterstützt. Man unterscheidet ein älteres, fixes MPS-Format und ein freies MPS-Format, das keine starren Spaltengrenzen, sondern nur eine beliebige Zahl von Blanks als Begrenzer zwischen den Daten einer Zeile erfor-

[1] [Forrest92] beschreibt, dass in der Pionierzeit der linearen Optimierung, Anfang der 50er-Jahre, sogar die Basisinverse bei jeder Iteration auf Lochkarte geschrieben und wieder eingelesen werden musste.

dert. Alle genannten Solver können mit beiden Formaten umgehen, wobei CPLEX sogar komprimierte MPS-Dateien schreiben und lesen kann.
Das MPS-Format ist aufgrund seiner Wurzeln aus der Lochkartenzeit in vielerlei Punkten für heutige Rechner nicht mehr optimal: Es enthält Datenredundanzen, beschränkt Namen auf acht Zeichen, ist wenig intuitiv lesbar und neigt zum Abschneiden vom Nachkommastellen, da Zahlen nicht binär, sondern als Zeichen geschrieben werden. Aus diesem Grund verfügen die meisten Solver über proprietäre, binäre oder zeichenbasierte Datenformate, die die besagten Probleme nicht aufweisen. Im Fall von MOPS ist das beispielsweise das *Triplet-Format* (vgl. [Suhl04, S. 72 ff.]), das folgende Vorteile gegenüber dem MPS-Format bietet:

- Redundanzfreiheit
- Lange Variablen- und Restriktionsnamen
- Namen alphanumerisch frei wählbar
- Intuitive Lesbarkeit für Menschen
- Einfache programmatische Handhabung (Parsing, Schreiben)

Ein weiteres Format ist das *LP-Format* (vgl. [Gollmer05, S. 15 ff.]), das in produktspezifischen Variationen von CPLEX, XPRESS und einigen anderen Solvern (z.B. LP_Solve, MOSEK) unterstützt wird. Sein Vorteil ist die leichte und intuitive Lesbarkeit, die der algebraischen Notation eines Modells recht nahe kommt. Das LP-Format eignet sich daher auch gut für das Debugging von Modellen.

Gemeinsame Funktionen

Lösungsdateien können in unterschiedlichen binären oder zeichenbasierten Formaten geschrieben werden kann. Weiterhin sind zur Lösungsanalyse in allen Bibliotheken Funktionen vorhandenen, die das Irreducibly Infeasible Set (IIS) eines unlösbaren Modells in eine Datei schreiben (zur IIS-Analyse vgl. [Guieu99]). LINDO kann auch das Irreducibly Unbounded Set (IUS) in ein File ausgeben.
Alle Bibliotheken besitzen Funktionen für „Optimierungs-Warmstarts“, wobei CPLEX und XPRESS neben der Basis auch eine MIP-Startdatei schreiben und einlesen können. Ferner lassen sich Branching-Prioritäten für Integervariablen aus Dateien einlesen. Eine weitere Funktion, die außer XPRESS alle untersuchen Solver bieten, ist das Einlesen von Parameterdateien, welche Einstellungen für Optimierungsparameter beinhalten. Schließlich können alle genannten Solver Logfiles ausgeben, was insbesondere zu Debugging-Zwecken und zur Überwachung des Optimierungslaufs hilfreich ist.

3.1.2.3 Modellaufbau und -modifikation

Gemeinsame Funktionen

Zu dieser Gruppe zählen zunächst Funktionen, um dem Modell Variablen und Restriktionen sowie Nonzeros hinzuzufügen. Dabei gibt es solver-spezifische Unterschiede, ob eine Zeile oder Spalte mit oder ohne den sie betreffenden Nonzeros eingefügt werden kann: CPLEX bietet Funktionen für beide Möglichkeiten, in XPRESS und LINDO müssen die Nonzeros mitgegeben werden. Ebenfalls können alle Bibliotheken einzelne oder mehrere Nonzeros setzen. Zur Übergabe eines kompletten Modells existieren in XPRESS und LINDO mehrere spezifische Funktionen (z.B. `XPRSloadlp`).

Das Einfügen von Special Ordered Sets (SOS, vgl. z.B. [Snyder84]), wird von allen Systemen unterstützt, indem für bereits angelegte Variablen die Zugehörigkeit zu einem SOS deklariert wird. Ebenfalls ist in allen genannten Bibliotheken das Hinzufügen, bzw. die Deklaration von Semi-Continuous Variablen (vgl. [Guéret02, S. 39]) möglich – etwa durch Funktionen wie `LSloadSemiContData` in LINDO. Semi-Continuous-Variablen und Special Ordered Sets können mit spezifischen Bibliotheksfunktionen modifiziert oder gelöscht werden.

Was die Modifikation eines Modells anbelangt, so haben alle Solver Funktionen zum Löschen einzelner oder mehrerer Variablen oder Restriktionen. Ebenso können einzelne Attribute von Variablen (Name, Typ, LB, UB, Zielfunktionskoeffizient) und Restriktionen (Name, Typ, RHS, LHS) geändert werden, wobei in der Regel das jeweilige Attribut eines unzusammenhängenden Blocks von Variablen oder Restriktionen mit einem Aufruf geändert werden kann. Alle untersuchten Solver bieten die Möglichkeit, einem Modell einen eigenen Namen zu geben bzw. diesen zu ändern, was z.B. zur Unterscheidung des Outputs vorteilhaft sein kann, wenn mit mehreren Modellen gleichzeitig gearbeitet wird. Selbstverständlich muss sich auch die Optimierungsrichtung ändern lassen, was bei CPLEX und LINDO über einen Parameter und bei XPRESS über den Aufruf unterschiedlicher Funktionen zum Start der Optimierung (`XPRSmaxim/XPRSminim`) geschieht. Schließlich sei noch erwähnt, dass die Solver Konstanten in der Zielfunktion erlauben, was eigentlich für die Optimierung überflüssig ist, jedoch in seltenen Fällen gebraucht wird, wenn z.B. die Solverbibliothek mit Modellierungssprachen, wie AMPL, gekoppelt werden soll, die ebenfalls Zielfunktionskonstanten erlauben.

Spezifische Funktionen

CPLEX bietet darüber hinaus noch spezifische Unterstützungsfunktionen für so genannte *Lazy Constraints*, also für Restriktionen, die nicht verletzt werden dürfen, deren Verletzung aber sehr unwahrscheinlich ist. Diese sind in einem MIP-Modell nicht Teil der Matrix der

relaxierten Modelle, sondern werden erst überprüft, nachdem eine Integer-Lösung gefunden wurde. Dieses CPLEX-spezifische Feature kann sich positiv auf die Rechenzeit auswirken, wenn die Lazy Constraints tatsächlich nicht verletzt werden, andernfalls müssten sie mit rechenzeitmäßig sehr teuren Operationen in die Matrix eingefügt werden.

3.1.2.4 Modellabfrage

Die Funktionen der Gruppe *Modellabfrage* dienen zum Auslesen von Modellinformationen nachdem das Modell erzeugt oder eingelesen wurde. Auch nach der Optimierung können in der Regel Modellinformationen ausgelesen werden, da die Solver das ursprüngliche Modell wiederherstellen, bzw. sichern, selbst wenn es durch Presolve und Optimierung intern stark verändert wurde.

Gemeinsame Funktionen

Alle untersuchten Solverbibliotheken enthalten Funktionen, mit denen die Attribute von Variablen (Name, Typ, LB, UB, Zielfunktionskoeffizient) sowie von Restriktionen (Name, Typ, RHS, LHS) ausgelesen werden können. CPLEX und XPRESS erlauben dabei, zusammenhängende Blöcke von Variablen und Restriktionen zu lesen, während LINDO nur einzelne Spalten bzw. Zeilen lesen kann. Nonzeros können zeilen- oder spaltenweise ausgelesen werden.

Für das Auslesen des Modellnamens haben CPLEX und XPLESS die expliziten Funktionen `CPXgetprobname` bzw. `XPRSgetprobname`, während die Vektorennamen von RHS, Bounds etc. in allen Bibliotheken über bestimmte Parameter geschrieben und gelesen werden. Zum Suchen des zu einem bestimmten Zeilen- oder Spaltennamen gehörigen Index, existieren in allen Libraries entsprechende Funktionen (z.B. `XPRSgetindex` in XPRESS). Ebenfalls haben alle Bibliotheken Funktionen zur Abfrage von Semi-Continuous-Variablen und SOS.

Das Auslesen der Modelldimensionen, d.h. der Anzahl der Zeilen, Spalten, Nonzeros, Integervariablen, Sets etc., geschieht recht uneinheitlich: Während CPLEX für sämtliche Werte gesonderte Funktionen in seiner Callable Library bereithält, erfolgt die Abfrage in XPRESS und LINDO über entsprechende Parameter.

Spezifische Funktionen

Eine LINDO-spezifische Besonderheit sind drei Funktionen zum Auffinden und zur Ausgabe von Blockstrukturen, mit denen eine Dekompositionsanalyse in Bezug auf Zeilen und/oder Spalten durchgeführt werden kann.

3.1.2.5 Lösungsabfrage und -analyse

Gemeinsame Funktionen

Was die Funktionalitäten zum Auslesen einer Lösung anbelangt, so verfügen alle untersuchten Solverbibliotheken über Funktionen, die die LP- und IP-Lösungswerte, die reduzierten Kosten und den Basisstatus von Variablen auslesen. Die in XPRESS und LINDO vorhandenen Funktionen liefern die entsprechenden Werte für alle vorhandenen Variablen und Restriktionen, CPLEX kann darüber hinaus auch selektiv zusammenhängende Blöcke auslesen. Lösungswerte, die für Restriktionen ausgelesen werden können, sind Activities, Dualwerte, Schlupfvariablen und Basisstatus.

CPLEX, XPRESS und LINDO verfügen ferner über Funktionen zum Suchen und Abfragen von Irreducibly Infeasible Sets (IIS) unlösbarer Modelle. Die LINDO Callable Library kann außerdem auch Irreducibly Unbounded Set (IUS) finden und auslesen. Funktionen zur Range-Analyse von Variablen- und Restriktionsbounds und von Zielfunktionskoeffizienten sind ebenfalls überall vorhanden.

Weiterhin existiert eine Vielzahl von Lösungsinformationswerten in Form von einzelnen Integer- oder Double-Werten, die bestimmte Attribute der Lösung oder des erfolgten Lösungsprozesses wiedergeben, beispielsweise die Anzahl der durchgeführten Simplex-Iterationen, die Anzahl der Knoten, der Lösungsstatus und Vieles mehr. Diese Werte werden in der Regel als Parameter über universelle Parameterfunktionen ausgelesen, die CPLEX Callable Library enthält darüber hinaus aber auch eine Anzahl von spezifischen Funktionen, die derartige Lösungsinformationswerte auslesen, wie z.B. `CPXgetmipitcnt` zur Abfrage der kumulierten Simplex-Iterationsanzahl im MIP oder `CPXgetnodecnt` für die Anzahl der im MIP untersuchten Knoten. Die Tabelle in Anhang 1 listet für CPLEX 26 Funktionen auf, die Lösungsinformationswerte wiedergeben, die auch in Form eines Parameters ausgelesen werden könnten.

Spezifische Funktionen

Ferner existiert noch eine Reihe weiterer, spezifischer Funktionen in den prozeduralen Bibliotheken von CPLEX und XPRESS, die eine erweiterte Lösungsanalyse unterstützen. So gibt es Funktionen, die bestimmte Maßzahlen für Rundungsfehler berechnen, Steepest Edge Norms ausgeben, ein Mapping von ursprünglichem und Presolved-Modell liefern und Einiges mehr. Diese Funktionalitäten sind stark auf die spezifischen Lösungsalgorithmen der jeweiligen Solver zugeschnitten und werden meist nur selten gebraucht.

3.1.2.6 Lösungsprozesssteuerung

In der Funktionsgruppe *Lösungsprozesssteuerung* befinden sich Funktionen, die den Optimierungsprozess in Gang setzen und lenkend in diesen eingreifen oder Informationen zur Steuerung des Lösungsprozesses liefern.

Gemeinsame Funktionen

Alle betrachteten Solverbibliotheken verfügen über Funktionen zum Start einer LP- oder IP-Optimierung, wobei CPLEX unterschiedliche Startfunktionen für die LP-Optimierung hat, je nachdem, welcher Algorithmus zur Lösung benutzt wird (`CPXprimopt` für Primalen Simplex, `CPXdualopt` für Dualen Simplex etc.). Weiterhin besitzen alle Solver Funktionen zum Setzen einer Startbasis. CPLEX und XPRESS können des Weiteren benutzerdefinierte Cuts festlegen und auch wieder löschen. Im Unterscheid zu CPLEX arbeitet XPRESS dabei mit einem so genannten *Cut Pool*, in dem benutzerdefinierte Cuts gespeichert werden und mittels bestimmter Bibliotheksfunktionen an den gewünschten Knoten aktiviert werden können. Ferner gibt es Funktionen zum expliziten Ausführen von BTRAN- und FTRAN-Operationen (zu BTRAN/FTRAN vgl. [Suhl90]).

Spezifische Funktionen

Die solverspezifischen Funktionen beziehen sich folglich bei XPRESS unter anderem auf das Management des Cut Pools (Cuts im Cut Pool speichern etc.). Außerdem besitzt XPRESS eigene Funktionen zur Steuerung des Branching durch das Setzen von Branching-Prioritäten, Directions und Pseudokosten. CPLEX besitzt ebenfalls eine Reihe von spezifischen Funktionen zur Lösungsprozesssteuerung, wie beispielsweise zum Herein- und Herauspivotieren von Variablen, zum Berechnen von Zeilen und Spalten der Basisinversen, zum expliziten Durchführen eines Presolves, sowie einige weitere Funktionen für eher selten gebrauchte Sonderfunktionen.

3.1.2.7 Parameterhandling

Alle Optimierer machen ausgiebigen Gebrauch von Parametern, teilweise auch „Attribute“ genannt, die eine Vielzahl von Einstellungen festlegen und Informationswerte liefern. Allein LINDO besitzt über 140 Parameter – Parameter für non-lineare Optimierung noch nicht mitgezählt. Man kann unterscheiden:

Inputparameter erlauben es dem Benutzer einer Optimierungsbibliothek Einstellungen festzulegen, deren Hauptzweck die Parametrisierung der Lösungsalgorithmen ist. Die getroffenen

Einstellungen haben, insbesondere bei MIP-Modellen, maßgeblichen Einfluss auf die Geschwindigkeit und die Qualität der Lösung. Die Auswirkung der Wahl bestimmter Input-Parameter (z.B. unterschiedliche Knotenauswahlstrategien etc.) auf die MIP-Leistungs–fähigkeit der hier betrachteten Optimierer CPLEX, XPRESS und LINDO wird beispielsweise in [AtaSav05] dargestellt.

Outputparameter liefern Informationen über das Modell, den Lösungsprozess oder die Lösung selbst. Hierzu zählen etwa die Modelldimensionen, die Anzahl der erfolgten Iterationen, die für die Optimierung benötigte Zeit und Vieles mehr.

Je nach Datentyp des zu setzenden oder auszulesenden Parameters existieren unterschiedliche Funktionen für Double-, Integer- und String-Parameter. LINDO besitzt darüber hinaus auch typunabhängige Parameterfunktionen, über die auch numerische Parameter mit den Funktionen `LSgetModelParameter` und `LSsetModelParameter` als Strings übergeben werden können.

3.1.2.8 Callbacks

Callback-Funktionen sind benutzerdefinierte Funktionen oder Prozeduren des die Solverbibliothek aufrufenden Programms, die zu bestimmten Ereignissen oder Zeitpunkten aus der Solverbibliothek heraus angesprungen werden. Nachdem also das den Optimierer einbettende Programm durch Start des Optimierungslaufs die Programmflusskontrolle an den Optimierer übergeben hat, ruft dieser bei Eintritt eines bestimmten Ereignisses eine Funktion des Benutzerprogramms auf, ohne aus der ursprünglichen Routine bereits zurückgekehrt zu sein.

Gemeinsame Funktionen

Alle hier betrachteten Optimierungsbibliothken unterstützten Callback-Funktionen, die bei einer ganzen Reihe von Ereignissen aufgerufen werden können: beispielsweise nach optimaler Lösung eines MIP-Subproblems, nach einer LP-Iteration, wenn eine bessere Integer-Lösung gefunden wurde, wenn Logging erfolgt usw. Insgesamt lassen sich 26 Ereignisse zählen, für die in den betrachteten Optimierungsbibliotheken Callbacks festgelegt werden können. Dabei gibt es jeweils gesonderte Bibliotheksfunktionen, um für ein bestimmtes Callback-Ereignis einen Funktionspointer auf eigenen Code zu setzen. Zum Beispiel setzt `CPXsetlpcallbackfunc` in CPLEX eine Callbackfunktion, die nach jeder LP-Iteration aufgerufen wird. LINDO besitzt lediglich Funktionen für drei Callback-Ereignisse und erlaubt ansonsten den Callback-Aufruf eigener Funktionen in bestimmten zeitlichen Intervallen.

Callback-Funktionen haben zwei Hauptanwendungsbereiche: Zum einen ermöglichen sie den Aufruf von Benutzerfunktionen zum Logging oder zur Ausgabe von Statusinformationen, zum anderen erlauben sie die gezielte Steuerung des Optimierungsprozesses, da sie beispielsweise die Implementation eigener MIP-Lösungsverfahren erlauben. Letzteres trifft allerdings nur auf CPLEX und teilweise auf XPRESS zu. Die primär zeitorientierten Callbacks in LINDO eigenen sich nur zur Ausgabe von Statusinformationen und Ähnlichem.

Spezifische Funktionen

Eine Besonderheit von CPLEX ist, dass es zum Auslesen bestimmter Lösungs- und Modellinformationen innerhalb von Callbackfunktionen gesonderte Funktionen gibt, die nur innerhalb von Callbackfunktionen angewendet werden dürfen. So liefert beispielsweise `CPXgetcallbackinfo` innerhalb eines Callbacks Zielfunktionswert, Iterationszahl und weitere Werte.

3.1.2.9 Fehlermeldungen und sonstige Funktionen

Die Funktionsgruppe *Fehlermeldungen* besteht in allen Solverbibliotheken nur aus einigen wenigen Funktionen: Die Funktionsaufrufe sind in den Callable Libraries in der Regel so konzipiert, dass entweder der Rückgabewert der Funktion oder eine Outputvariable einen Statuscode enthält, der Auskunft über den Ausführungserfolg der Funktion gibt. Diese numerischen Returncodes können vom aufrufenden Programm aus direkt ausgewertet werden. Zuweilen ist es aber auch hilfreich, eine explizite Fehlermeldung in Stringform zu bekommen, hierzu existieren in allen Bibliotheken geeignete Funktionen.

In die letzte Funktionsgruppe, den *Sonstigen Funktionen,* fallen Funktionen zur Abfrage von Versionsinformationen sowie zum Auslesen von Lizenzdaten. Eine Besonderheit bei CPLEX sind diverse Validierungsfunktionen, die die Argumente anderer Funktionen vorab auf Richtigkeit prüfen. So prüft z.B. `CPXcheckaddcols` die Argumente der Funktion `CPXaddcols` und liefert ggf. einen Fehlercode.

3.2 Objektorientierte Optimierungssystemschnittstellen

In diesem Unterkapitel wird der Stand der Technik in Bezug auf objektorientierte Solverbibliotheken dargestellt. Die Grundprinzipien objektbezogener Programmierung werden in Kapitel 4 behandelt. Einen knappen Einstieg in objektorientierte Programmierparadigmen mit dem Hintergrund Optimierungssystemschnittstellen findet man darüber hinaus in [Fink02a].

Bei den objektorientierten Bibliotheken lassen sich zwei Gruppen unterscheiden: Klassen- und Komponentenbibliotheken. Komponentenbibliotheken bestehen in Gegensatz zu Klassenbibliotheken aus unmittelbar ausführbaren Binärmodulen (vgl. [Fink02a]). Im Gegensatz zu [Szyperski02, S. 35 ff], der zu den Komponenten auch prozedurale Binärbibliotheken (etwa prozedurale DLLs) zählt, sollen aufgrund der hier vorgenommenen Gliederung nur objektorientierte Komponentenbibliotheken betrachtet werden. Optimierungssystemschnittstellen in Form von Klassenbibliotheken sind hauptsächlich für C++ und Java vorhanden, während sie als Komponentenbibliotheken unter Windows für das COM- und das .NET-Framework existieren.
In [Fourer05] sind insgesamt 18 Optimierungssysteme mit objektorientierten Schnittstellen aufgeführt, von denen aber nur wenige hier in Betracht kommen, da die dort aufgeführten Systeme teils integrierte Modellierungssysteme sind (z.B. AIMMS), oder weil es sich nicht um echte Klassensysteme handelt. Letzteres ist z.B. für LINDO der Fall, wo die prozeduralen Interfaces lediglich mit Deklarationen für die entsprechenden objektorientierten Programmiersprachen (VB.NET, C#) versehen wurden, so dass sie aus objektorientierten Umgebungen angesprochen werden können. Die beiden wohl umfangreichsten objektorientierten Bibliotheken sind die *Concert Technology Library* von ILOG ([ILOG06a]) und die *BCL* von Dash Optimization ([Dash06b]). Beide sind als Klassenbibliotheken für C++ und Java, sowie als Komponentenbibliothek für .NET vorhanden und sollen im Folgenden einander gegenüber gestellt werden.

3.2.1 Klassenbibliotheken

3.2.1.1 BCL

Die *Builder Component Library (BCL)* von *Dash Optimization* ist eine objektorientierte Klassenbibliothek zur Benutzung mit dem Optimierer XPRESS für die Programmiersprachen C, C++, Java und Visual Basic (vgl. [Dash06b]). Anders, als der Name vermuten lässt, handelt es sich bei dieser Bibliothek jedoch nicht um eine Komponentenbibliothek, sondern vielmehr um eine Klassen- und Funktionsbibliothek. Für die Benutzung unter C++ existieren Header-Files für die Deklaration der Klassen, deren Objektcode während des Builds aus einer statischen Bibliothek hinzugefügt wird. Dies ist die normale, für Klassenbibliotheken typische Prozedur, bei der Binärkomponenten (COM/.NET) nicht zum Einsatz kommen. Gleiches gilt für die C-Version der BCL, nur dass mangels Objektorientierung in C lediglich Funktionen und keine Klassen eingebunden werden.

Neben der statischen Bibliothek existiert noch eine DLL-Version der BCL, die insbesondere für die Benutzung aus Visual Basic wichtig ist, da in Visual Basic keine statische Einbindung möglich ist. Die VB-Version der BCL weist einen prozeduralen, der C-Version ähnlichen Charakter auf. Im Folgenden wird nur die C++-Variante der BCL, die intern sehr eng mit der C-Variante verknüpft ist, näher beschrieben.

Die C++-Version der XPRESS Builder Component Library besteht aus insgesamt zehn einzelnen Klassen, die untereinander nicht in Vererbungszusammenhängen stehen (s. Anhang 2). Diese zehn Klassen kann man in zwei Gruppen aufteilen: Die Gruppe der Kern-Klassen enthält sechs Klassen, die für die Modellierung der meisten Probleme notwendig sind:

- **`XPRB`** enthält globale Funktionalitäten zur Initialisierung, Lizenzüberprüfung, Versionsinformation und Systemzeit.
- **`XPRBprob`** repräsentiert ein Modell und bildet daher das Zentrum der BCL. Es existieren Methoden zum Erzeugen von Variablen, Restriktionen, Index-Sets, Cuts und SOS, sowie zum Löschen von Restriktionen, Cuts und SOS. Das Modell und die Basis können in Dateien gespeichert und wieder ausgelesen werden. Die Klasse `XPRBprob` dient auch zum Start des Optimierungslaufs und zur Abfrage von Statusparametern und Zielfunktionswerten während und nach der Optimierung. Für unlösbare Modelle steht eine Methode zur Analyse und Ausgabe des IIS zur Verfügung.
- **`XPRBvar`** dient der Darstellung von Entscheidungsvariablen. Es gibt Methoden zum Setzen und Auslesen der typischen Variablenattribute wie Ober- und Untergrenze, Typ und Name. Es können nicht nur kontinuierliche und Integer-Variablen, sondern auch Partial-Integer und Semi-Continuous-Variablen abgebildet werden. Weiterhin können Lösungswerte, reduzierte Kosten und Range-Informationen abgefragt werden.
- **`XPRBctr`** repräsentiert eine Restriktion und enthält Methoden zum Lesen und Schreiben von Restriktionsattributen wie Name, RHS, Typ, und Range. Lösungsinformationen wie Activities, Dualwerte und Slacks sind auslesbar, und es können Restriktionen als Cuts definiert werden. Des Weiteren gibt es Methoden zum Hinzufügen und Löschen eines Terms (Koeffizient * Variable). Die Möglichkeit, in C++ Operatoren zu überladen, wurde ausgenutzt, indem die Operatoren =, += und -= für das Zuweisen, Hinzufügen und Löschen von linearen Ausdrücken überladen wurden, so dass Formulierungen wie `myConstraint += myLinearExpression` möglich werden.
- **`XPRBlinExpr`** dient der Formulierung linearer Ausdrücke. Dabei wird die Hauptfunktionalität durch die Überladung der Operatoren +, -, *, =, += und -= erreicht, was

Formulierungen nahe an algebraischen Ausdrucksweisen ermöglicht, wie etwa: `myLinExpr = myVar1 * myValue1 + myVar2 * myValue2.`

- **`XPRBerror`** zur Abfrage numerischer Fehlercodes und Text-Fehlermeldungen.

Eine zweite Gruppe enthält zusätzliche Klassen, deren Funktionalitäten die Kernklassen ergänzen, die aber nur für besondere Modellierungsfälle gebraucht werden:

- **`XPRBindexSet`** unterstützt das Arbeiten mit geordneten Stringmengen, die als Indizes dienen. Indexnamen können dynamisch hinzugefügt und ausgelesen werden, wozu es sowohl Methoden als auch überladene Operatoren gibt.
- **`XPRBsos`** dient der Abbildung eines Special Ordered Sets vom Typ 1 oder 2. Elemente lassen sich hinzufügen, gewichten und löschen, sowie Branching Directions definieren.
- **`XPRBcut`** stellt einen Cut in einem MIP-Modell dar. Dazu verfügt die Klasse über ähnliche Methoden wie die Restriktionsklasse. Es können Terme dem Cut hinzugefügt und gelöscht werden, der Cuttyp (<=, =, =>) lässt sich setzen und auslesen, und Operatoren sind ähnlich wie bei Restriktionen überladen, so dass sich Cuts entsprechend natürlich formulieren lassen.
- **`XPRBbasis`** dient der Repräsentation einer Basis.

Die C++-Version der BCL bildet mit den beschriebenen Klassen eine relativ dünne objektorientierte Kapsel um die darunter liegende prozedurale C-Version der Bibliothek, da die überwiegende Zahl der Methoden der C++-Klassen lediglich analoge Funktionen der C-Bibliotheksversion aufrufen. Die C-Version der BCL darf nicht mit dem in Unterkapitel 3.1 beschriebenen C-Interface des XPRESS Solvers (Callable Library) verwechselt werden. Auch wenn es eine große Anzahl von funktionalen Überschneidungen mit dem Solver Interface gibt, so stellt die C-Version der BCL doch eine eigene Bibliothek dar, deren primäre Aufgabe die Erleichterung der Modellierung aus der Programmiersprache C ist, wohingegen das Solver Interface zwar kompletten Zugriff auf alle Features des Optimierers gibt, die Modellgenerierung aber nur in Form eines klassischen Matrixgenerators unterstützt. Zur syntaktischen Unterscheidung beginnen alle Funktionsnamen des XPRESS Solver Interface mit „XPRS“ und alle Funktionsnamen der C-Version der BCL mit „XPRB“.

Eine parallele Benutzung der BCL und der prozeduralen XPRESS-Optimiererschnittstelle ist möglich und in manchen Fällen sogar unumgänglich, da nicht alle Features des Optimierers über die BCL zugänglich sind. Bei paralleler Nutzung beider Schnittstellen dürfen allerdings nur solche Interfacefunktionen des Optimierers genutzt werden, die nicht die eigentliche Mo-

dellmatrix betreffen, da diese von der BCL verwaltet wird. Dieses Verhältnis von BCL und Solverinterface ist schematisch in Abbildung 8 dargestellt. Das Abstraktionsniveau der Modellierung ist bei der Benutzung der BCL höher als bei Benutzung der Optimiererschnittstelle, da in der BCL in der Regel mit höheren Konstrukten, wie z.B. linearen Objektausdrücken, gearbeitet wird, während die Callable Library des XPRESS-Solvers nur mit der Matrixdarstellung arbeitet. Dieser Unterschied tritt insbesondere in der C++-Version der BCL im Fall der überladenen Operatoren zu Tage.

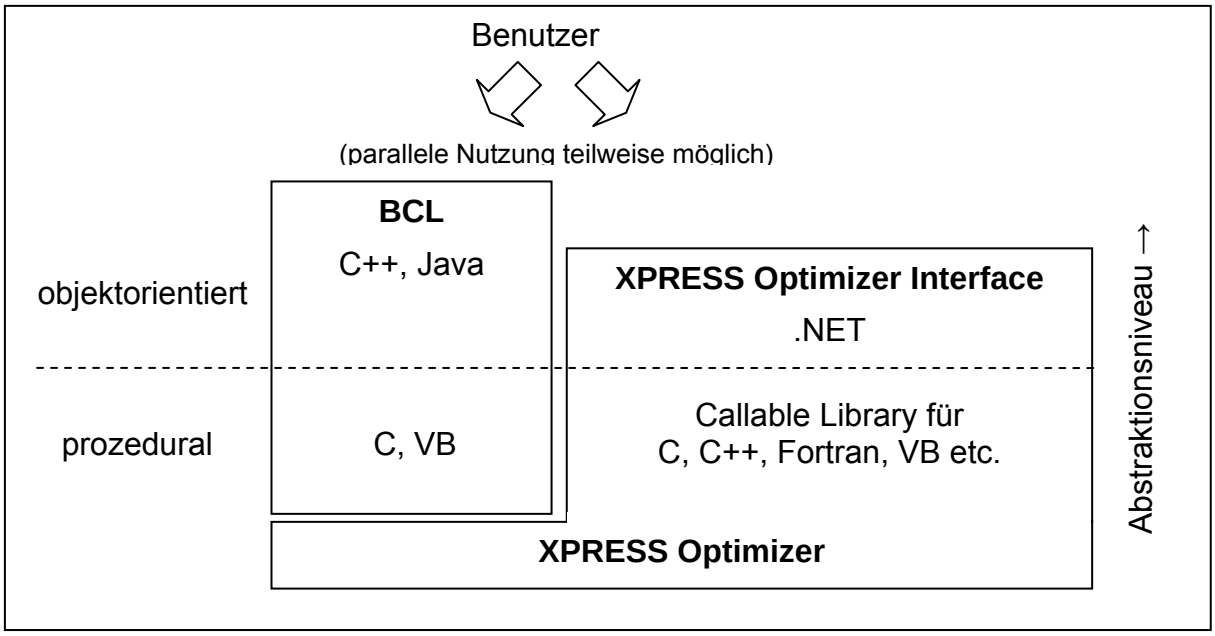

Abbildung 8: BCL und XPRESS Optimizer Interface

3.2.1.2 ILOG Concert Technology Library

Die ILOG Concert Technology ist eine innovative objektorientierte Schnittstellenbibliothek, die die Einbindung von CPLEX und von ILOG Solver durch ein gemeinsames Interfacesystem ermöglicht. Während CPLEX lineare, gemischt-ganzzahlige und quadratische Programmierung unterstützt, dient der ILOG Solver zur Lösung von Constraint Programming-Problemen (vgl. [Lustig01] zum Unterschied zwischen mathematischer Programmierung und Constraint Programming). Beide Optimierungsengines verfügen über eigene Schnittstellenbibliotheken: Solver besitzt eine C++-Schnittstelle und CPLEX kann über die in Unterkapitel 3.1 beschriebene Callable Library angesprochen werden. Neben diesen nativen Schnittstellen bildet Concert Technology ein gemeinsames Interfacesystem, in dem der Benutzer mathematische Modelle oder kombinatorische Constraint-Modelle erstellen kann, wobei die zu Grunde liegenden Optimierungsengines zu einem hohen Grad transparent bleiben. Die an [Clarke06] angelehnte Abbildung 9 schematisiert diesen Zusammenhang und zeigt darüber hinaus die

Solver-Add-Ons *Scheduler*, *Dispatcher* und *Configurator* sowie das ebenfalls CPLEX und Solver benutzende *OPL-Studio*.

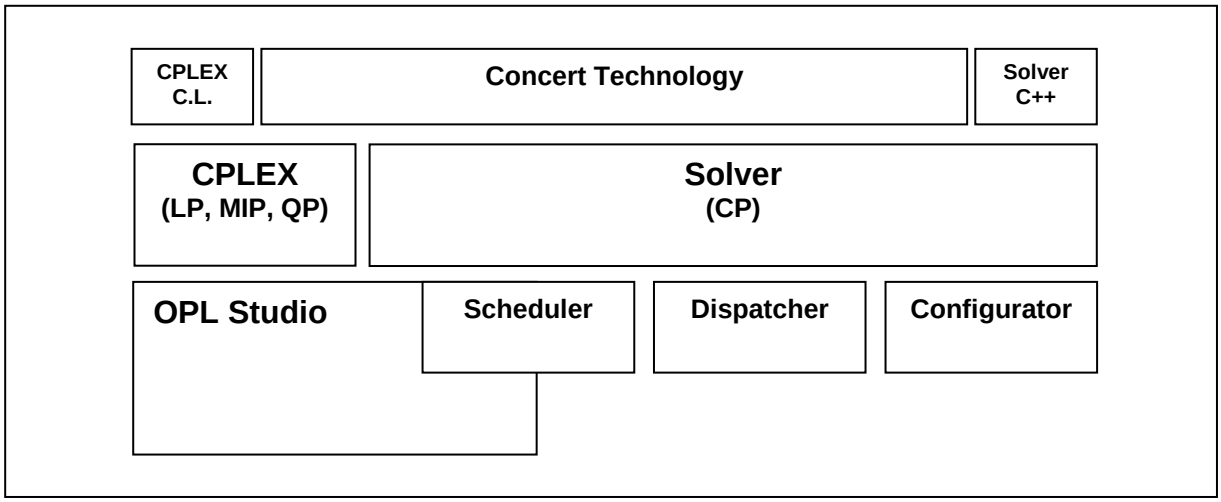

Abbildung 9: Schnittstellen ILOG CPLEX und ILOG Solver

Im Folgenden werden Klassen und Struktur der Concert Technology detailliert beschrieben, wobei der Fokus auf den Teilen der Concert Technology Library liegt, die lineare Optimierung mit CPLEX unterstützen; Teilbereiche, die sich auf nicht-lineare Optimierung oder Constraint Programming beziehen, werden ausgelassen. Die Concert Library ist in Versionen für C++, Java und .NET vorhanden. Da sich die Java-Version, abgesehen von Operatorüberladungen, nicht wesentlich von der C++ Version unterscheidet, wird hier zunächst die C++-Version beschrieben und die komponentenorientierte .NET-Version später nur kurz angerissen. Als Quelle dienen vor allem die entsprechenden Referenzhandbücher ([ILOG06c] sowie ferner [ILOG06b]). Anhang 3 enthält eine Darstellung der relevanten Klassenstruktur der Concert Library für den LP-/MIP-Bereich.

Die ILOG Concert Technology-Bibliothek besteht aus einer Vielzahl von zum Teil verschachtelten Klassen, die mitunter in kaskadierten Vererbungszusammenhängen stehen. Hinzu kommt eine Reihe von Makros und Definitionen, die zusammen mit den Klassen ein komplexes, in sich geschlossenes Rahmenwerk zur objektorientierten Modellierung bilden. Dabei sind die meisten Klassen als Paar vorhanden, bestehend aus einer Handle-Klasse und einer Implementationsklasse. Die Handle-Klasse ist die Klasse, mit der der Benutzer primär arbeitet; sie enthält einen Pointer auf die entsprechende Implementationsklasse, um die sie eine nur dünne Kapsel bildet. Vorteil dieses Konstrukts ist, dass zum einen mehrere Handle-Klassen auf eine Implementationsklasse zeigen können und der Benutzer zum anderen mit sehr

schlanken Objekten in Bezug auf den Ressourcenverbrauch arbeitet und Konstruktor- und Destruktor-Mechanismen effizienter ablaufen können.

`IloEnv`, `IloAlgorithm`, `IloCplex`

Im Mittelpunkt der Bibliothek stehen die Klassen `IloEnv`, `IloAlgorithm` und `IloCplex`:

Eine Umgebung wird repräsentiert durch die Klasse `IloEnv`, die zum einen Methoden für bestimmte globale Funktionalitäten zur Verfügung stellt und zum anderen wichtige Aufgaben des Speichermanagements übernimmt. Zu den globalen Funktionalitäten zählen die Verwaltung der Streams für Outputs, Fehler und Warnungen sowie Abfragemöglichkeiten für Versionsinformationen, Systemzeit, Speicherverbrauch und ferner einige Methoden zum Setzen und Abfragen einiger weiterer globaler Einstellungen.

Die beiden Optimierungsengines, CPLEX und ILOG Solver werden durch die Klassen `IloCplex` und `IloSolver` dargestellt, die `IloAlgorithm` als gemeinsame Elternklasse besitzen. Nach seiner Generierung wird ein Modell je nach Typ einer dieser Engines zur Lösung übergeben („extrahiert“). Ein Objekt der Klasse `IloCplex` oder `IloSolver` stellt also nach Extraktion sowohl den Algorithmus als auch das Modell dar. Die gemeinsame Elternklasse `IloAlgorithm` umfasst dabei Funktionalitäten, die für beide Optimierer gemeinsam gelten, wie z.B. Abfrage von Zielfunktionswerten, Modellstatus und Lösungswerten für Variablen, ebenso wie Methoden zum Start des Optimierungslaufs, zum Extrahieren oder Löschen eines Modells und zur Verwaltung von umgebungsspezifischen Streamzeigern.

`IloCplex` ist die bei Weitem mächtigste Klasse, bezogen auf den Funktionsumfang. Sie ermöglicht die Abfrage einer Reihe von Modellattributen, wie Anzahl der Spalten, Zeilen, Nonzeros, Integer-, Semi-Integer- und Semi-Continuous-Variablen, sowie von Lösungsattributen, wie Lösungsstatus, Anzahl der untersuchten Knoten, der Iterationen, der verbleibenden Knoten, der Crossover-Operationen etc. Weiterhin können die diversen Lösungswerte des Modells ausgelesen werden: Variablenwerte, reduzierte Kosten, Dualwerte und Slacks. Der Lösungsprozess des Modells kann insbesondere bei MIP-Modellen vom Benutzer durch verschiedene Wege beeinflusst werden. So ist es beispielsweise möglich, die Klasse `IloCplex` anzuweisen, zuvor definierte Callbacks während des Lösungsprozesses aufzurufen, Branching Prioritäten und Branching Directions lassen sich setzen und auslesen, ebenso können Cuts und Lazy Constraints dem Modell hinzugefügt und auch wieder gelöscht werden. Neben dem normalen Lösungsprozess kann über die Klasse `IloCplex` auch ein separater Presolve-Lauf gestartet werden, oder der Optimierer kann über die Methode `feasOpt` angewiesen werden,

unter Lockerung von Restriktionen auf jeden Fall eine zulässige Lösung zu finden. Methoden zur Sensitivitätsanalyse für Variablenbounds, RHS und Ranges stehen nach dem Lösungslauf zur Verfügung. Weiterhin erfolgt die Parameterverwaltung über die Klasse `IloCplex`, indem diese Methoden zum Setzen und Auslesen einzelner Parameter und ganzer Sets von Parametern anbietet. Schließlich kann `IloCplex` auch eine Reihe von Modellinformationen wie Basis, Parameter, Lösung, MIP-Start etc. in Dateien ausgeben.

`IloExtractable` und abgeleitete Klassen

Die nächste große Gruppe von Klassen der Concert Library sind die von `IloExtractable` abgeleiteten Klassen. Dabei handelt es sich um Klassen, deren Instanzen direkt oder indirekt nach `IloCplex` oder `IloSolver` extrahierbar, also übertragbar sind. Insbesondere ist in dieser Gruppe die Klasse `IloModel` von Bedeutung, die ein Modell vor Extraktion in ein Engine-Objekt repräsentiert. Andere wichtige Klassen dienen der Darstellung von Zielfunktion (`IloObjective`), kontinuierlichen Variablen (`IloNumVar`), Integervariablen (`IloIntVar`) und Restriktionen (`IloConstraint`). Daneben gibt es Klassen für Binärvariablen (`IloBoolVar`) und für semi-kontinuierliche Variablen (`IloSemiContVar`) sowie die Klasse `IloConversion`, die zur Typänderung einer Variablen dient. Eine wichtige Rolle beim Aufbau von linearen Ausdrücken für Restriktionen spielt die Klasse `IloExpr` mit ihrer Elternklasse `IloNumExpr`, deren Hauptfunktionalität durch die teilweise mehrfach überladenen Operatoren +=, -=, *=, /= entsteht, so dass im Zusammenspiel mit den globalen, d.h. außerhalb von Klassen überladenen Operatoren +, -, *, / Ausdrücke wie beispielsweise

```
myIloExpr = myIloNumVar + 5 * myotherIloNumVar;   oder
myIloExpr += 4 * myIloNumVar;
```

möglich werden. Aus diesen linearen Ausdrücken können Restriktionen der Klasse `IloConstraint` zusammengesetzt werden, die anschließend dem Modell `(IloModel)` hinzugefügt werden müssen. Zum Beispiel:

```
myIloConstraint = (myIloExpr + myotherIloExpr <= 4711);
myIloModel.add(myIloConstraint);
```

Dieser Konstruktionsmechanismus für Restriktionen ermöglicht C++-Code, der syntaktisch relativ nahe an die algebraische Notation eines linearen Modells herankommt und die Stärke der Concert Library ausmacht.

Zu den extrahierbaren Klassen zählt auch eine Reihe von `IloConstraint` abgeleitete Klassen. Dies sind zum einen Klassen wie `IloSOS1` und `IloSOS2`, zur Abbildung von Special Ordered Sets, oder `IloRange` zur Darstellung von Range-Restriktionen und zum

anderen Klassen, wie `IloIfThen`, `IloNot`, `IloOr` und `IloAnd`, die eine interessante Doppelrolle zwischen Constraint Programming und mathematischer Optimierung spielen. Diese Klassen dienen eigentlich der Formulierung von logischen Zusammenhängen für das Constraint Programming, wie etwa *„Restriktion1 oder Restriktion2 müssen erfüllt sein.“* (`IloOr`) oder *„Wenn Restriktion1 erfüllt ist, muss auch Restriktion2 erfüllt sein.“* (`IloIfThen`). Sie können aber in der Regel auch nach `IloCplex` extrahiert werden, wobei die Concert Technology dann die notwendige Binärvariablenmodellierung automatisch und für den Benutzer transparent vornimmt.

Weitere Klassen

Neben den vorangehend beschriebenen Klassen, die das Kernstück der Modellierung mit Concert Technology bilden, gibt es noch eine Anzahl weiterer Klassen für besondere Funktionalitäten. Dabei sei zunächst der Bereich der Callbacks genannt: Für die diversen Punkte, an denen ein Callback aufgerufen werden kann, existieren jeweils eigene Implementationsklassen, die innere Klassen von `IloCplex` sind. So gibt es beispielsweise Klassen für Callbacks nach Simplex-Iterationen (`IloCplex::SimplexCallbackI`) oder während des Presolves (`IloCplex::PresolveCallbackI`), ebenso wie für unterschiedliche Punkte im MIP-Lösungsprozess (`IloCplex::BranchCallbackI` etc.). Da die Instanzierung einer Callback-Klasse fünf Schritte erfordert, gibt es in der C++-Version der Concert Library eine Reihe von Makros zur Vereinfachung der Benutzung.

Die Fehlerbehandlung erfolgt in der Concert-Klassenbibliothek nicht wie in der prozeduralen Callable Library über Returncodes, sondern über Ausnahmebehandlung (Exceptions), wobei die potentiell Fehler verursachenden Aufrufe und Operationen in einen `try`-Block eingeschlossen werden in der Form

```
try
     {Methodenaufrufe von Concert Objekten}
catch (IloException& myException)
     {Ausnahmebehandlung}
```

Für unterschiedliche Fehlersituationen existieren insgesamt 13 Exception-Klassen, die aber allesamt, direkt oder indirekt, von `IloException` abgeleitet sind.

Ein System von dynamischen, d.h. Größen veränderbaren und Speicher selbst verwaltenden Array-Klassen bietet für die meisten Modellierungsobjekte einen passenden Array als Container. So gibt es Arrays für Fließkomma-, Integer- und Binärwerte (`IloNumArray`, `IloIntArray`, `IloBoolArray`), für kontinuierliche, ganzzahlige und binäre Variablen

(`IloNumVarArray, IloIntVarArray, IloBoolVararray`) sowie für Restriktionen, Special Ordered Sets und lineare Ausdrücke. Alle diese Klassen sind von der Templateklasse `IloArray` abgeleitet und überladen den Operator [] für Zugriffe auf einzelne Feldelemente.
Schließlich besitzt die Concert Library noch einige Klassen, die hier nur kurz erwähnt seien: Es sind die Klassen zur Unterstützung von Mengen (`IloAnySet` etc.) und dazugehörigen Iteratoren zum Durchlaufen dieser Mengen, weiterhin Hilfsklassen zum Auslesen von Textdateien (`IloCsvReader` etc.) und zum Schreiben und Lesen von XML-Files.

3.2.1.3 Vergleich BCL und Concert Technology

BCL und Concert Technology Library sind in Bezug auf Architektur und Ausrichtung sehr unterschiedlich, so dass eine direkte Identifikation analoger Funktionen – wie in Unterkapitel 3.1 für prozedurale Bibliotheken erfolgt – weder möglich noch sinnvoll ist.
Der wohl wichtigste Unterscheid zwischen beiden Klassenbibliotheken liegt in der unterschiedlichen Zielsetzung: Die BCL soll die Modellgenerierung durch Bereitstellung einer objektorientierten Schicht unterstützen. Die Concert Library dient zwar ebenfalls diesem Zweck, fungiert aber gleichzeitig auch als gemeinschaftliches Interface für *CPLEX* und die Constraint Programming Engine *ILOG Solver*. Diese Doppelrolle bedingt Klassen (z.B. `IloAllDiff`, `IloNot` etc.), die nur in CP-Modellen eingesetzt werden und daher in der lediglich mathematische Optimierung unterstützenden BCL fehlen.
Weitere Unterschiede sind, dass die Klassen der Concert Library in komplexen Vererbungszusammenhängen miteinander stehen, teilweise innere Klassen besitzen und von Templates Gebrauch machen. Die BCL hingegen besteht aus unabhängigen, sehr schlanken C++-Klassen, die meist lediglich Funktionen der C-Version der BCL kapseln. Auch besitzt die BCL nicht das zweischichtige Prinzip der Handle-Klassen, das in der Concert Library Verwendung findet. Die Anzahl der Klassen in der Concert Library ist höher und ihr Zusammenspiel ist erheblich differenzierter, aber auch komplizierter.
Einige Klassen, die Grundelemente eines LP-/MIP-Modells wie Variablen, Restriktionen und Zielfunktion repräsentieren, sind in beiden Bibliotheken vorhanden, jedoch entsprechen sie sich nur in sehr groben Zügen. So besitzt die BCL beispielsweise eine einzige Klasse `XPRBvar` zur Darstellung einer Strukturvariablen, die Concert Library hat hingegen mehrere unabhängige Klassen für kontinuierliche Variablen (`IloNumVar`), Integervariablen (`IloIntVar`), Binärvariablen (`IloBoolVar`) und Semi-Continuous Variablen, die teilweise auch über spezielle Methoden für Constraint Programming verfügen (z.B.

`IloIntvar.setPossibleValues`). Unterschiede gibt es auch bei Restriktionen: In der BCL existiert eine Klasse `XPRBctr` für alle Arten von Restriktionen, in der Concert Library werden neben normalen Restriktionen (`IloRange` für ≤, ≥, =) auch z.B. Special Ordered Sets als Restriktion aufgefasst, da sie von der gemeinsamen Basisklasse `IloConstraint` abgeleitet sind.
Obwohl einige Mechanismen, wie die Definition von Restriktionen mittels überladener Operatoren, in beiden Bibliotheken ähnlich sind, dominieren dennoch insgesamt die Unterschiede.

3.2.2 Komponentenbibliotheken

Die derzeitigen State-of-the-Art-Optimierungssysteme unterstützen von den am Markt befindlichen Komponententechnologien (z.B. COM, .NET, VCL, JavaBeans etc.) im Wesentlichen nur die .NET Plattform. Nicht-Microsoft-Komponententechnologien, die ohnehin nur einen sehr kleinen Anteil am Komponentenmarkt haben ([Szyperski02, S. 25-26]), werden praktisch nicht unterstützt, und auch für das ansonsten sehr populäre und marktdominante COM findet sich unter den verbreiteten Optimierungssystemen lediglich in der *MPL Library* ([Maximal06]) eine Anbindung. Wie bei der vorangegangenen Beschreibung des Stands der Technik von Klassenbibliotheken, sollen auch hier die Optimierungssysteme von Dash und ILOG herangezogen werden. Anschließend wird noch kurz auf die MPL als Beispiel für eine COM-basierte Komponentenbibliothek eingegangen. Als Quellen dienen die jeweiligen Produktdokumentationen ([ILOG06c], [Dash06e], [Maximal06b]).

3.2.2.1 XPRESS-MP .NET

Das Optimierungspaket XPRESS-MP bietet in der aktuellen Version 2006A auf zweierlei Art eine komponentenorientierte Anbindung für .NET und seine Sprachen (C#, VB.NET, J# etc.): Zum einen existiert eine .NET-Version der Einbindungsbibliothek für die Modellierungssprache MOSEL (dazu mehr in Kapitel 3.4), zum anderen wurde der XPRESS-Optimizer mit einem komponentenorientierten .NET-Wrapper versehen. Die Assembly `xprsdn` enthält die Bestandteile dieses Wrappers. Dies sind zunächst die Klassen `XPRS` und `XPRSprob`, wobei Erstere vor allem der Initialisierung und dem Lizenzmanagement dient, während `XPRSprob` den eigentlichen Kern der Komponentenbibliothek ausmacht. Für fast alle Funktionen der prozeduralen Bibliothek des XPRESS-Optimizers gibt es analoge Methoden und Eigenschaften in `XPRSprob`, die sich größtenteils entsprechen, jedoch vom Benennungsschema auf die syntaktischen Konventionen des .NET-Frameworks angepasst wurden. Die Callbacks der prozeduralen Bibliothek wurden auf den für .NET typischen Event-Delegate-Mechanismus um-

gestellt, und auf Returncodes wurde zu Gunsten eines strukturierten Exception Handlings verzichtet. Zusammenfassend kann gesagt werden, dass es sich bei der .NET-Komponentenbibliothek des XPRESS-Optimizers eher um einen Wrapper, als um eine neue und eigenständige Bibliothek handelt, da die Architektur der prozeduralen Optimiererbibliothek unter der komponentenorientierten Schicht noch gut erkennbar ist.

3.2.2.2 Concert Komponentenbibliothek für .NET

Die CPLEX/Concert-Komponentenbibliothek für .NET unterscheidet sich hingegen deutlicher von ihrem Klassenbibliothekspendant. Sie besteht aus den beiden .NET-Assemblies `ILO.Concert` und `ILOG.CPLEX`, die auf zwei DLLs verteilt sind. Wie auch die Concert-Klassenbibliothek, so bildet auch die .NET-Komponentenbibliothek ein gemeinsames Interface für Modelle der mathematischen Programmierung, die mit CPLEX gelöst werden, sowie Constraint Programming-Modellen, die mit ILOG Solver gelöst werden. Somit bleibt auch im .NET-Bereich die Idee der Trennung von Modell und Algorithmus erhalten, auch wenn die Ausgestaltung der Bibliotheken im Detail unterschiedlich ist. Herausragendes Merkmal der Concert-Komponentenbibliothek ist die Verwendung von Interfaces. Interfaces definieren ein Set von Methoden und Eigenschaften und schreiben so einen Kontrakt fest. Die hinter diesen Kontrakten stehenden Funktionalitäten werden durch Klassen realisiert, die die jeweiligen Interfaces implementieren. Die für .NET typische Verwendung von Interfaces ist der wichtigste Unterscheid zu der in 3.2.1.2 beschrieben C++ Klassenbibliothek. Insgesamt existieren über 30 Interfaces, die untereinander in vielfältigem Vererbungszusammenhang stehen. Das wichtigste Modellierungsinterface ist `IModeler`, das über sein Basisinterface `IModel` das Hinzufügen und Entfernen von Modellierungsobjekten ermöglicht und darüber hinaus eine Vielzahl eigener Methoden bereitstellt, z.B. zum Hinzufügen von Restriktionen durch Angabe der RHS und eines numerischen Ausdrucks (`INumExpr`). Von `IModeler` abgeleitet ist das Interface `IMPModeler`, das erweiterte Funktionen für mathematische Programmierung enthält (z. B. Zugriff auf Matrixdarstellung, spaltenweise Modellgenerierung etc.). Weiterhin existieren noch Interfaces für das Handling von Restriktionen (`IRange`, `IConstraint`), Zielfunktionen (`IObjective`), Variablen (`INumVar`, `IIntVar` etc.). Die meisten Interfaces sind von den Basisinterfaces `ICopyable` und `IAddable` abgeleitet, die Funktionen zum Kopieren und zum Hinzufügen zu Modellen in die abgeleiteten Interfaces einfügen.
Die Funktionalitäten der Interfaces werden hauptsächlich in den Klassen des Namspaces `ILOG.CPLEX` implementiert. Die zentrale Modellierungsklasse `Cplex` implementiert unter anderem die Interfaces `IModel`, `IModeler` und `IMPModeler`, über die ein großer Teil der

Modellgenerierung erfolgt. Die Klasse `Cplex` enthält darüber hinaus aber auch eine Vielzahl eigener Methoden und Eigenschaften. So existieren beispielsweise Methoden zur Sensitivitätsanalyse, für das Einlesen von Modellen, Parametern und Basislösungen aus Files, zum Einfügen von Cuts und Lazy Constraints und zum Auslesen von Lösungsinformationen. Die Modelldimensionen (Anzahl von Variablen, Spalten, Nonzeros) sind – ebenso wie eine Reihe weiterer Werte – als Eigenschaften der Klasse `Cplex` abfragbar. Neben der Hauptklassen `Cplex` gibt es noch eine Reihe weiterer Klassen, etwa zur Repräsentation eines IIS oder zur so genannten Goal-Programmierung (Details s. [ILOG06c]). Weiterhin sind über 20 Enumerationen als Klassen modelliert: Die Klasse `Cplex.Algorithm` enthält beispielsweise eine Auflistung von möglichen Lösungsalgorithmen (Barrier, Simplex etc.). Im Gegensatz zur Klasse `IloAlgorithm` aus der C++/Java-Klassenbibliothek enthält die .NET Klasse `Cplex.Algorithm` jedoch keinerlei algorithmus-spezifische Implementation – sie ist lediglich eine Aufzählung, ähnlich einer C++-Enumeration.

Eine weitere Gruppe in der .NET-Komponentenbibliothek bilden die Callback-Klassen, die fast genau dem Callback-Klassensystem der C++/Java Concert Library entsprechen. Um während des Optimierungslaufs benutzerspezifischen Code zur Statusabfrage oder zur Beeinflussung der Optimierung ablaufen zu lassen, gibt es eine Reihe von Callback-Klassen, die als Elternklassen für neue benutzerspezifische Callback-Klassen dienen. Der Benutzer wählt die Callback-Klasse aus, die dem Event oder Punkt entspricht, an dem er seinen Code ausführen möchte (z.B. `Cplex.SimplexCallback` für Callbacks nach jeder Simplexiteration) und erstellt dann eine neue von dieser Callback-Klasse abgeleitete Klasse, in der er die virtuelle Funktion `Main()` mit seinem Code überlädt. Zur Laufzeit wird ein `Cplex`-Objekt von der Existenz des benutzerdefinierten Callbacks über die Methode `use()` unterrichtet, und `Cplex` ruft dann zum gewünschten Punkt die Main-Methode des Callback-Objekts auf.

Weiterhin besitzt die Concert .NET-Komponentenbibliothek Exception-Klassen, die die .NET-typischen Exception-Mechanismen unterstützen. Von ihren Pendants in den Concert C++/Java-Klassenbibliotheken unterscheiden sie sich insbesondere in Bezug auf die internen Vererbungszusammenhänge. Wie auch die Callable Library und die Klassenbibliotheken unterstützt die CPLEX/Concert-.NET-Bibliothek ferner logische Restriktionen (And, Or, If Then), die angeben, ob eine Restriktion aktiviert ist oder nicht. Deren Generierung erfolgt für den Benutzer transparent, wobei intern keine klassische Big-M-Formulierung in Verbindung mit Binärvariablen benutzt wird, sondern numerisch robustere *Indicator Constraints* ([ILOG06c, S. 313-314]).

Schließlich sei noch erwähnt, dass die .NET-Komponentenbibliothek – anders als die C++-Klassenbibliothek – keinen Gebrauch von überladenen Opertoren macht, obwohl dies für einen Teil der .NET-Sprachen (C#, Managed C++) möglich wäre.

3.2.2.3 OptiMax 2000

Als Dritte und Letzte soll die MPL OptiMax 2000 Komponentenbibliothek ([Maximal06]) hier noch kurz erwähnt werden, die eine der ganz wenigen COM-basierten Optimierungsbibliotheken ist. Die OptiMax 2000-Bibliothek dient weniger der eigenständigen Modellgenerierung, sondern vor allem der Einbindung von Modellen der Modellgenerierungssprache MPL in ein Anwenderprogramm. Typischerweise wird ein MPL-Modell zunächst in der grafischen Benutzeroberfläche *MLP for Windows* entwickelt und anschließend die resultierende Textdatei mit der Modellformulierung mithilfe der COM-Modellierungsobjekte der MPL-Komponentenbibliothek in eine benutzererstellte Anwendung eingebunden. Zur Laufzeit der Anwendung kann das so geladene MPL-Modell gelöst werden, wobei die Verwendung diverser Solver (CPLEX, XPRESS u.a.) möglich ist. Zur Abfrage der Optimierungsergebnisse steht eine Reihe von COM-Objekten (Variable, Restriktion, Set, Matrix etc.) zur Verfügung. Die MPL-Komponentenbibliothek ist grundsätzlich in allen Programmiersprachen verwendbar, die ActiveX-/COM-Automatisierung unterstützen.

3.3 Integrierte Modellierungssysteme

Unter einem integrierten Modellierungssystem versteht man ein Softwaresystem, das zumindest einige der Schritte des Modellierungszyklus unterstützt, wobei „integriert“ in diesem Zusammenhang bedeutet, dass die einzelnen Bestandteile des Systems zwar prinzipiell auch einzeln benutzbar sind, jedoch durch ihre Zusammenfügung einen größeren Zweckerfüllungsgrad erreichen ([Geoffrion89, S. 3]). Abbildung 10 stellt den Modellierungszyklus nach [Geoffrion89] und den weiteren dort genannten Quellen dar. Die einzelnen Schritte müssen nicht streng sequentiell erfolgen und können teils auch übersprungen werden.

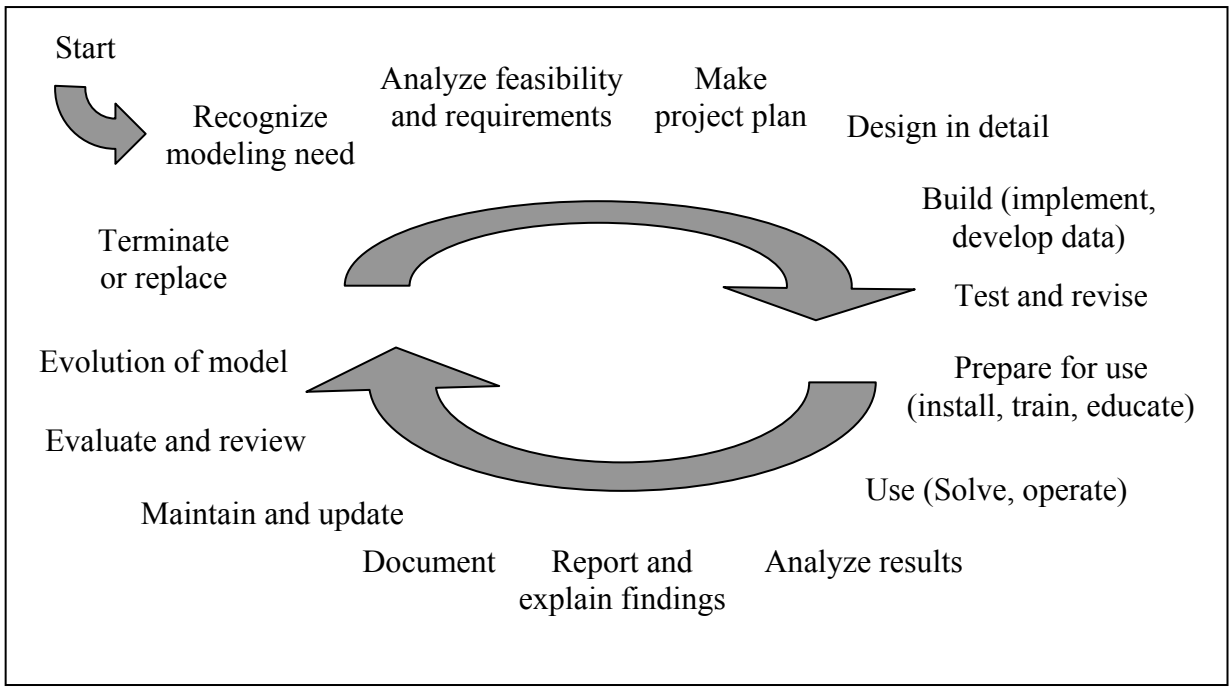

Abbildung 10: Modellierungszyklus

Das Konzept der Integration kann bei einem Modellierungssystem folgende Kategorien umfassen (vgl. [Geoffrion89]):

- Modelle
- Solver
- Utilities

Modellintegration bedeutet nicht nur, dass mehrere Instanzen eines Modells oder mehrerer Modelle einer Klasse zusammengefasst und verwaltet werden können, sondern es ist auch denkbar, dass ein Modellierungssystem die Kombination mehrerer Modellklassen ermöglicht. So könnte für bestimmte Fragestellungen beispielsweise eine Verbindung von MIP-Modellen mit Simulationsmodellen sinnvoll sein, wobei das Modellierungssystem die integrative Instanz in Bezug auf Modellverzahnungen und Datenmanagement wäre. Ebenso kann ein Modellierungssystem hinsichtlich unterschiedlicher Solver integrierend wirken, indem es die Anbindung von Solvern verschiedener Herkunft für den Benutzer transparent macht. Als drittes bezieht sich die Integrationsidee auf die Einbeziehung und kooperative Steuerung verschiedener Utilities, wie etwa Werkzeuge für grafisches und textbezogenes Reporting, Datenmanagement, statistische (Daten-)Analyse, Modelleditoren, Texteditoren, etc.
Die wichtigste Aufgabe von Modellierungssystemen ist neben der Modellverwaltung vor allem die Unterstützung der Modellerstellung, wozu die Integration einer algebraischen Modellierungssprache in fast allen modernen Systemen den wohl größten Beitrag leistet.

Im Folgenden sollen einige state-of-the-art-Modellierungsysteme in Bezug auf ihre Unterstützung der Schritte des Modellierungszyklus und in Bezug auf ihre integrativen Konzepte untersucht werden.

3.3.1 ILOG OPL Development Studio

Zur Integration seiner Optimierungsengine CPLEX mit der Modellierungssprache OPL bietet ILOG das mit einer grafischen Benutzeroberfläche ausgestattete Optimierungssystem *OPL Development Studio* in der momentan aktuellen Version 5.0 an ([ILOG06d], [ILOG06e]). Dabei beschränkt man sich auf die Kombination „OPL + CPLEX“ – andere Modellierungssprachen oder andere Solver werden nicht unterstützt. Wie in anderen Modellgenerierungssprachen und -systemen auch, wird eine strenge Trennung von Modell und Daten verfolgt. Modelle, Daten und sonstige Einstellungen sind auf unterschiedliche Dateien verteilt und werden in Projekten organisiert. Grafische Browser ermöglichen die baumartige Darstellung der Modellelemente, Daten und Lösungsergebnisse. Ebenfalls durch grafische Oberflächenelemente unterstützt ist die Parametrisierung der Optimierungsengine CPLEX. Außerdem existiert ein Analyzer zum Auffinden von Modellkonflikten und ein Zeit- und Ressourcen-Profiler. Weiterhin bietet OPL Development Studio Anbindung an die Datenbanken Oracle, DB2, MS-SQL sowie ODBC.
Zwar werden nicht alle Schritte des o.g. Modellierungszyklus in OPL Development Studio vollständig abgebildet, aber die Unterstützung umfasst die Kernelemente der Modellierung: Modellerstellung, Datenhandling, Lösungsprozesssteuerung, Ergebnisanalyse und Reporting. Das Integrative im oben genannten Sinne bezieht sich nicht auf die Integration unterschiedlicher Solver, sondern vielmehr auf die Verbindung von Solver, Modellierungssprache und grafischer Werkzeugunterstützung.

3.3.2 XPRESS IVE

Die integrierte grafische Optimierungsumgebung *XPRESS IVE* von Dash Optimization ([Dash06f]) ähnelt dem OPL Development Studio, nur dass die Modellierungssprache MOSEL und der XPRESS Optimizer anstelle von OPL und CPLEX kombiniert werden. Unterstützt werden neben LP- und MIP-Modellen auch non-lineare und stochastische Programmierung sowie Constraint Programming, so dass eine Integration über mehrere Modellklassen besteht. Es existieren mehrere grafische Tools zur Darstellung von Modellelementen, Daten, Modellmatrix, B&B Baum und Lösungsinformationen. Ebenfalls gilt der Grundsatz der Trennung von Modell und Daten, die in der Regel auf unterschiedliche Dateien verteilt sind.

In Bezug auf den Modellierungszyklus werden die gleichen Kernschritte wie bei OPL Development Studio unterstützt. Besonderheit ist allerdings eine Reihe von Wizards, die für unterschiedliche Modellierungsschritte wie Dateneingabe, Modellformulierung, Tuning, grafischer Output, Debugging etc. Hilfe bereitstellen, indem sie den notwendigen MOSEL-Code, ausgehend von Formularfeldeingaben, erstellen. MOSEL-Modelle können, wie OPL-Modelle auch, über Einbindungsbibliotheken in benutzerdefinierte Programme (C, Java, VB etc.) eingebettet werden. Die dazu notwendigen Anweisungen kann die XPRESS IVE über ihre Deployment Wizards automatisch generieren.

3.3.3 AIMMS

AIMMS (*Advanced Integrated Multi-dimensional Modeling Software*) ist ein sehr hoch integriertes Modellierungssystem, das auf einer eigenen Modellierungssprache und einer umfangreichen Konstruktionsumgebung für grafische Benutzeroberflächen beruht. Ein Systemüberblick findet sich u.a. auf der AIMMS-Website [Paragon06] und in [Bisschop04].

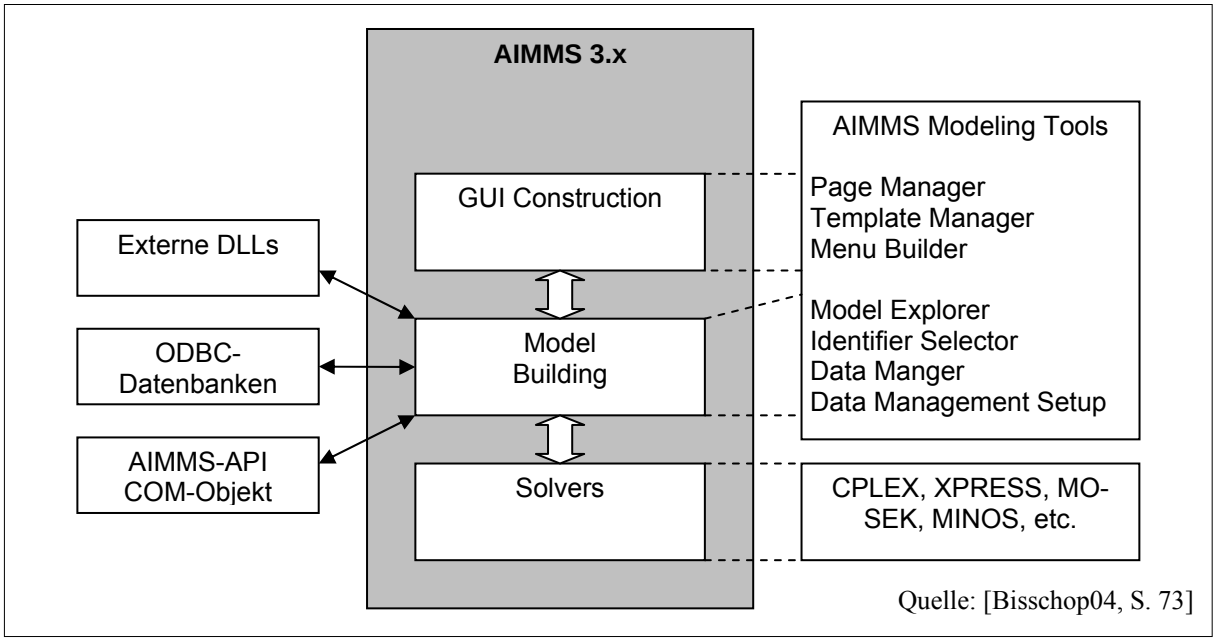

Abbildung 11: AIMMS-Systemüberblick

Abbildung 11 zeigt die Hauptbestandteile von AIMMS: *GUI Constructor*, *Model Builder* und *Solver*. Die Fähigkeit, eigene grafische Benutzeroberflächen zu erstellen, unterscheidet AIMMS von anderen integrierten Modellierungsumgebungen. Es steht eine Anzahl von eignen Oberflächenelementen zur Verfügung, die speziell auf die Erfordernisse eines Optimierungssystems angepasst sind. So gibt es neben GUI-Standardelementen, wie Buttons, Textboxen etc., auch spezielle Grafikcontrols, wie z.B. Network- oder Gantt-Charts, die ohne weite-

ren Programmieraufwand mit Datenreihen oder anderen Modellbestandteilen verbunden werden können. Bei Änderung der Ausgangsdaten aktualisieren sich die Steuerelemente automatisch und können teilweise auch Benutzereingaben entgegennehmen, so dass eine bidirektionale Synchronisation von Steuerelementen und Datenbasis entsteht. Weiterhin werden Menüerstellung und GUI-Templates vom *GUI Constructor* unterstützt.

Der *Model Builder* ist das Kernstück von AIMMS. Aus Benutzersicht unterscheidet er sich von anderen integrieren Modellierungssystemen dadurch, dass er menübasiert ist und keine Freitexteingabe einer Modellierungssprache erfordert. Dies reduziert auf der einen Seite Fehlermöglichkeiten, insbesondere syntaktischer Art, bringt aber auf der anderen Seite einen gewissen Mangel an Überblick mit sich. Um dies auszugleichen, gibt es Viewer (*Identifier Selector*) zur gegenüberstellenden Ansicht von Modellierungselementen. Modelle werden in AIMMS grundsätzlich als Auflistung von Modellierungsobjekten (Restriktionen, Variablen, Sets, Parameter etc.) in Form eines flachen Baumes dargestellt. Ein weiteres Alleinstellungsmerkmal von AIMMS ist das fallbezogene Datenmanagement, das Versionierung und Szenariorechnungen erleichtert. Dies wird ermöglicht durch die Strukturierung der Modelldaten in Form von Cases durch den *Data Manager* und das *Data Management Setup*.

Das Modellierungssystem AIMMS ist im Gegensatz zu OPL Development Studio und XPRESS IVE nicht an einen herstellerspezifischen Solver gekoppelt und bietet Anschluss an eine Reihe von state-of-the-art-Solvern wie CPLEX, XPRESS, MINOS, MOSEK und anderen. Neben linearen und gemischt-ganzzahligen können auch nicht-lineare und Netzwerk-Modelle gelöst werden.

Die Modellierungssprache AIMMS ist nicht rein deklaratorisch, sondern kann um benutzerdefinierte prozedurale Funktionen erweitert werden. Darüber hinaus besteht auch die Möglichkeit, Prozeduren und Funktionen aus externen DLLs aufzurufen, was den potenziellen Funktionalitätsbereich erheblich erweitert. Außerdem existieren – wie in anderen integrierten Modellierungssystemen auch – Schnittstellen zu diversen Datenbanken via ODBC. Weiterhin gibt es eine AIMMS-API in Form einer kleineren COM-Bibliothek, die es ermöglicht, AIMMS in externe, benutzerdefinierte Programme einzubinden. Das Vorgehen ist dabei typischerweise so, dass ein Modell in der Entwicklungsumgebung erstellt und getestet wird und dieses Modell anschließend von einer externen Anwendung über die COM-Schnittstellen der AIMMS-API aus einer Datei eingelesen wird. Auch die Modelldaten können so eingelesen und verändert werden. Die COM-Schnittstelle dient also nicht der Neuerstellung eines Modells, sondern dem Laden, Parametrisieren und Lösen eines bestehenden Modells. Ähnliche Mechanismen zur Einbindung gibt es auch in anderen Modellierungssprachen und –systemen,

wie etwa in OPL, XPRESS und MPL, auch wenn die dort existierenden Einbettungsschnittstellen im Detail verschieden sind.

AIMMS unterscheidet sich nicht nur durch sein weit reichendes Konzept zur Unterstützung integrierter Anwendungsentwicklung von anderen Modellierungssystemen, sondern auch durch einige weitere Features, die hier lediglich kurz erwähnt werden sollen (Details in [Bisschop04, S 84 ff.]): Modelldaten sind häufig mit einer bestimmten Maßeineinheit (z.B. Meter, Kilogramm etc.) versehen, deren Einbeziehung in die Modellierung oftmals inhaltsvollere Ausdrücke erlaubt und Fehler bei der Verknüpfung inkompatibler Maßeinheiten offenkundig werden lässt. AIMMS unterstützt die Modellierung mit Maßeinheiten durch umfangreiche Funktionen zur Konvertierung von Maßzahlen und zur Prüfung der Maßeinheiten-Konsistenz von Ausdrücken. Weiterhin besitzt AIMMS für die Modellierung von zeitbezogenen Größen spezielle Konzepte (*rolling horizons, calendars, timetables* etc.). Abschließend seien noch die mit einem Multi-Agent-Ansatz implementierten Features zur verteilten Modellierung und Optimierung erwähnt.

Von den hier untersuchten Modellierungssystemen ist AIMMS das am höchsten integrierte in Bezug auf die Anbindung unterschiedlicher Solver und Einbeziehung verschiedenster Utilities (Reporting, grafische Darstellung etc.). Auch wenn der Modellierungszyklus nicht komplett abgedeckt wird, so bedingt der hohe Integrierungsgrad doch eine Unterstützung aller wesentlichen Schritte des Modellierungszyklus, die teilweise über OPL Development Studio und XPRESS IVE hinausgeht.

3.4 Modellierungssprachen

Modellierungssprachen spielen eine zentrale Rolle, denn zum einen bilden sie neben den Solver Engines den Kern aller modernen integrierten Modellierungssysteme, zum anderen sind viele Modellierungssprachen über Einbettungsbibliotheken auch innerhalb eigenerstellter Anwendungen einsetzbar. Wie [Ciriani02] beschreibt, ging die Entwicklung von proprietären, koeffizientenorientierten Eingabeformaten der 50er- und 60er-Jahre über das MPS-Format zunächst in den frühen 80er-Jahren zu Matrixgeneratorsprachen wie DATAFORM, GAMMA und OMNI. Mit steigender Rechnerleistungsfähigkeit und mit wachsenden Ansprüchen der Benutzer folgten die auch heute noch verbreiteten Modellierungssprachen GAMS (1987, [Brooke88]), AMPL (1990, [Fourer90]) und LINGO (1990, [Schrage97]). Die jüngste Entwicklung ab Ende der 90er-Jahre war die immer stärkere Anreicherung der im Kern deklarativen Modellierungssprachen mit prozeduralen Elementen, um Datentransformationen und Ein- und Ausgabeaufgaben zu erleichtern und vor allem, um in den Lösungsprozess steuernd ein-

greifen zu können. Beispiele sind OPL/OPL-Script, MOSEL und teilweise auch AMPL. Daneben wurden mit steigender Popularität von Open Source auch im Bereich der Modellierungssprachen derartige Projekte verfolgt. Hier sind vor allem das ein Subset von AMPL darstellende GNU MathProg ([GNU06]) und ZIMPL ([Koch04], [Koch05]) zu nennen.
Die idealtypischen Features algebraischer Modellierungssprachen sind nach [Schichl04] und [Kallrath04b] die Folgenden:

- Syntax: Syntaktische Nähe zur algebraischen Notation, komplette sprachliche Abdeckung aller Features der potenziell benutzbaren Solver, freie Wahl von Variablen- und sonstigen Namen.
- Anbindung möglichst vielfältiger Solver (LP, MIP, NLP, MNLP, CP etc.).
- Offenheit bzgl. Datenanbindung (Files, Datenbanken, SAP, etc.)
- Plattformunabhängigkeit
- Unabhängigkeit von Modellstruktur und Modelldaten
- Effizientes Indexing, mit Indexsets, Subsets und darauf operierenden Funktionen
- Unterstützung der Infeasibility-Analyse (z.B. Ausgabe von IIS etc.)
- Skalierbarkeit bzgl. Performance und Ressourcenverbrauch: Lineare Performanceentwicklung und Ressourcenbeanspruchung auch bei Modellen mit Hunderttausenden oder Millionen von Objekten. Die Modellgenerierungszeit sollte erheblich kleiner sein als die Modellösungszeit.
- Deployment: Einbettungsmöglichkeiten in Anwendungen
- Versionsmanagement
- Verteilte Entwicklung und Optimierung

Im Folgenden sollen die Modellierungssprachen AMPL, OPL und MOSEL vergleichend dargestellt werden. Wie auch in den anderen Teilen dieser Arbeit, beschränkt sich die Betrachtung dabei auf die LP/MIP-Modellierung. Im Sinne eines „Vor-die-Klammer-Ziehens“ werden zunächst die in allen drei Sprachen gemeinsamen oder ähnlichen Elemente beschrieben und danach auf Besonderheiten der einzelnen Produkte eingegangen. Hierfür wurden im Wesentlichen folgende Quellen herangezogen: [Fourer03a] für AMPL, [vHentenr99], [ILOG06f] für OPL und [Colombani02], [Ciriani03], [Dash06d] für MOSEL. Eine umfassende und aktuelle Darstellung – auch weiterer Modellierungssprachen – findet man in [Kallrath04]. Ein schon etwas älterer Vergleich diverser Modellgenerierungssprachen ist [Steiger93], dort fehlen freilich die in der Zwischenzeit hinzugekommenen Features gegenwärtiger Sprachversionen. [Kristjansson04] gibt ebenfalls einen kurzen Vergleich von Modellierungssprachen an-

hand einer Liste von zehn Kriterien (Indexing, Memory Management, Robustness etc.). Dabei wird allerdings kaum auf Details bestehender Sprachimplementationen eingegangen, und syntaktische und semantische Unterschiede bleiben ganz außen vor.

3.4.1 Gemeinsame Sprachelemente

Die hier betrachteten Modellierungssprachen besitzen eine Reihe von gleichen oder ähnlichen Sprachelementen, die teilweise auch in anderen Modellierungssprachen (LINGO, MPL etc.) in ähnlicher Form existieren und sich häufig nur in Bezug auf Syntax oder kleinere Implementierungsdetails unterscheiden. Ähnlich wie in Kapitel 3.1 für Optimierungsbibliotheken geschehen, könnte man diesen gemeinsamen Sprachkern als „Common State-of-the-Art“ bezeichnen.

Bevor auf die gemeinsamen Sprachelemente im Detail eingegangen wird, soll die Struktur und typische Syntax der zu vergleichenden Modellierungssprachen anhand eines sehr einfachen Beispielmodells dargestellt werden (Tabelle 2):

AMPL

```
var x1 integer >= 0;
var x2 integer >= 0;

maximize Profit: 5*x1 + 20*x2;

working_hours: 3*x1 + 2*x2 <= 160 ;
material_avail: x1 + 3*x2 <= 200 ;

solve;
display Profit;
display x1;
display x2;
```

OPL

```
dvar int+ x1;
dvar int+ x2;

maximize 5*x1 + 20*x2;

subject to
{
   3*x1 + 2*x2 <= 160 ;
   x1 + 3*x2 <= 200 ;
}

execute
{
   writeln("Profit: ",cplex.getObjValue());
   writeln("Product 1: ", x1);
   writeln("Product 2: ", x2);
}
```

MOSEL

```
model Production
       uses "mmxprs"              ! Use Xpress-Optimizer for solving
       declarations
              x1,x2: mpvar        ! Decision variables
       end-declarations

       3*x1 + 2*x2 <= 160         ! Constraint: limit on working hours
       x1 + 3*x2 <= 200           ! Constraint: raw material availability

       x1 is integer; x2 is integer     ! Integrality constraints

       maximize(5*x1 + 20*x2)           ! Objective: maximize profit

       writeln("Solution: ", getobjval)        ! Print objective value
       writeln("Product 1:", getsol(x1))       ! Print solution variables
       writeln("Product 2:", getsol(x2))
end-model
```

Tabelle 2: Codebeispiele AMPL, OPL, MOSEL

Das Modell ist ein simples, eigentlich selbsterklärendes Produktionsplanungsproblem, bei dem der Profit aus der Produktion von zwei Gütern maximiert werden soll, wobei Arbeitszeit- und Materialrestriktionen zu beachten sind (entnommen aus [Colombani02, S. 2]). Zunächst fällt der unterschiedliche Grad der Strukturierung auf: Während das AMPL-Modell keine explizite syntaktische Untergliederung aufweist, sind die beiden anderen Sprachen stärker strukturiert. MOSEL trennt beispielsweise die Variabelndeklarationen syntaktisch klar vom Rest des Modells. Auch das Modell selbst wird als Block aufgefasst und in die Schlüsselworte `model` und `end-model` eingeschlossen. Ähnliches gilt für OPL, das wie die Programmiersprachen C++ und Java eine Blocksyntax mit geschwungenen Klammern benutzt. Alle drei Sprachen sind nicht rein deklarativ, sondern enthalten auch imperative Elemente - im Beispiel die Ausgabeanweisungen. Darüber hinaus können auch andere Programmiersprachen typische Strukturen, wie etwa Schleifen und If-Then-Anweisungen, genutzt werden. Eine Besonderheit von OPL ist die Trennung von deklarativen und imperativen Modellteilen, weshalb die Ausgabeanweisungen in einen `execute`-Block eingeschlossen werden müssen. Die prozeduralen Statements sind in OPL eine eigene Scriptsprache, *OPL Script* (vergl [vHentenr00]), die mittlerweile *ILOG Script* genannt wird und eine Implementation des ECMA-262 Standards (bekannt als JavaScript) darstellt.
Die übrigen Unterschiede der Sprachen sind – soweit sie dieses simple Beispielmodell zeigt – eher syntaktischer Natur.

Einfache Sets und Indizes

Sets (Mengen) sind ein zentrales Element aller Modellierungssprachen. Einfache Sets können in AMPL, OPL und MOSEL aus numerischen Werten und Strings bestehen. OPL und MOSEL erfordern bei der Deklaration eine explizite Typangabe, während AMPL den Typ auto-

matisch setzt. Sets bestehen aus unterschiedlichen Elementen, es kann keine doppelten Elemente innerhalb eines Sets geben. Die Initialisierung mit Mengenelementen kann im eigentlichen Modellcode erfolgen, sollte aber aufgrund der logischen Trennung von Modell und Daten besser im Datenbereich vorgenommen werden. Eine besondere Art von Sets sind Ranges, also Abfolgen von diskreten Werten in einem bestimmten Intervall, wobei auch Schrittweiten gewählt werden können. Ranges werden in den Modellierungssprachen in der Form `Start..Ende` dargestellt und spielen eine wichtige Rolle bei Indizierungsausdrücken. Ranges sind effizienter implementiert als echte Sets. Alle betrachteten Sprachen verfügen über Mengenoperatoren für Vereinigungsmenge, Schnittmenge, Differenzmenge etc. AMPL und OPL benutzen hierzu Schlüsselwörter (z.B. `union`), MOSEL hingegen sieht die Verwendung von Operatoren (+, -, *) vor.

AMPL
1: set MySet;
2: set MySet = {"bands", "coils", "plate"};
3: set YEARS = 1990 .. 2020;
4: set U := A union B;
5: param MyParam {MySet};
6: subject to MyRestr {i in MySet}: …
OPL
1: {float} MySet;
2: {string} MySet = {"bands", "coils", "plate"};
3: {int} YEARS = asSet(1990..2020);
4: {int} U = A union B;
5: int MyArray[MySet];
6: forall(i in MySet) …
MOSEL
1: MySet: set of real
2: MySet = {"bands", "coils", "plate"}
3: YEARS = 1990..2020
4: U = A + B oder auch z.B. U = {1,2,3} + {4,5,6}
5: MyArray: array(MySet) of integer
6: forall(i in MySet) …

Tabelle 3: Syntaxbeispiele Sets

Sets dienen insbesondere der Indizierung von Datenarrays, Variablen und Restriktionen. Tabelle 3 zeigt in den jeweiligen Zeilen 5 und 6 einfache Beispiele für eindimensionale Indizierungen. In den Zeilen 5 wird ein Datenarray mit einem zuvor definierten und gefüllten Set indiziert. Die jeweiligen Zeilen 6 zeigen, wie ein Set benutzt wird, um eine indizierte Gruppe von Restriktionen zu erzeugen. Abgesehen von syntaktischen Verschiedenheiten funktionieren derartige einfache Indizierungen in allen betrachteten Sprachen nach dem gleichen Prinzip.

Skalare Daten und Datenfelder

Datenvariablen, in manchen Modellierungssprachen auch „Parameter“ genannt, enthalten Ausgangsdaten, die aus einer vom Modell getrennten Datendatei oder aus einer externen Datenbank gefüllt werden, wobei das Auffüllen normalerweise während der Modellgenerierung ohne Zutun des Benutzers geschieht. Datenvariablen können als Datenarrays ein- oder mehrdimensional sein und sind dann über Sets oder Ranges indiziert.

AMPL
```
1: param MyPar;
2: param MyArray {1..10};
3: param MyArray {MySet1, MySet2};
4: param MyPar >= 0 integer;
5: param MyPar {i in 1..10}
         >= if t = 1 then 0 else MyPar [t-1];
```
OPL
```
1: int MyInt; float MyFloat; string MyString;
2: int MyIntArray [1..10];
3: int MYIntArray [MySet1, MySet2];
4: int MyInt = ...;
   assert MyInt >= 0;
5: int MyInt [i in 1..10]= ...;
   assert forall(i in 2..10) MyInt[1] >= MyInt[i-1];
```
MOSEL
```
1: MyInt: integer;  MyFloat: real;  MyString: string;
2: MyArray: array(1..10) of integer;
3: MyArray: array(MySet1, MySet2) of integer;
4: MyInt: integer     ! im Deklarationsblock
   assert(MyInt >= 0) ! ausserhalb des Deklarationsblocks
5: MyInt: array (1..10) of integer
   forall(i in 2..10) assert( MyInt(i) >= MyInt(i-1))
```

Tabelle 4: Syntaxbeispiele Datenvariabeln und -arrays

Tabelle 4 zeigt in den Zeilen 1 die Deklaration einfacher Datenvariablen bzw. Parameter sowie in den Zeilen 2 und 3 die eines ein- und zweidimensionalen Arrays.

Um Fehler in den Ausgangsdaten zu erkennen, bevor diese sich als Fehler im Modell manifestieren können, besitzen die untersuchten Modellierungssprachen die Möglichkeit, Prüfbedingungen für Datenvariablen und Arrays zu definieren. In OPL und MOSEL geschieht dies mit dem `assert`-Befehl, in AMPL lassen sich solche Bedingungen direkt bei der Variablendeklaration definieren (Tabelle 4, Zeilen 4). Häufig handelt es sich dabei um Nichtnegativitätsbedingungen, oder um Einschränkungen auf bestimmte Wertebereiche. Neben solchen einfachen Bedingungen können aber auch komplexere Ausdrücke formuliert werden. Die Zeilen 5 in Tabelle 4 zeigen beispielsweise eine Bedingung, die sicherstellt, dass die Werte eines Arrays stetig größer werden, was etwa bei der Überprüfung eines Vektors von kumulierten Werten sinnvoll sein kann.

Selbstverständlich können Datenvariablen auch das Ergebnis eines berechneten Ausdrucks annehmen (*Computed Parameters*), was wichtig ist, da die Ausgangsdaten häufig nicht in

einer unmittelbar im Modell verwendbaren Form vorliegen, etwa weil Maßeinheiten oder Größenordnungen umgerechnet werden müssen. Ein weiteres wichtiges gemeinsames Feature der Modellierungssprachen sind dünn besetzte Arrays, auf die im folgenden Abschnitt eingegangen wird.

Erweiterte Set-Funktionalitäten

Neben einfachen Aufzählungen oder Ranges lassen sich Sets auch aus komplexeren Ausdrücken erzeugen. Dabei wird ein über eine Grundmenge laufender Iterator bestimmt, auf den eine Bedingung angewendet wird und der zusätzlich für die Zielmenge durch einen Ausdruck modifiziert werden kann. Diese Art von Mengen wird in OPL *Generic Sets* genannt, existiert mit minimalen Unterschieden jedoch auch in AMPL und MOSEL. Die jeweils ersten Zeilen in Tabelle 5 zeigen ein Beispiel für Generic Sets, in dem der Iterator über die Range 1..4 läuft, wobei Element 2 ausgeschlossen wird und der Iteratorwert in OPL und MOSEL noch durch den Ausdruck `i*3` modifiziert wird.

AMPL
`1: set s = { i in 1..4 : i != 2};  # ergibt {1,3,4}`
`2: set S ordered = {1,2,3};`
`   param i = first(S);`
`3: set S = {(1,2),(3,4)};`
OPL
`1: {int} S = {i*3 | I in 1..4: i != 2};  // ergibt {3,9,12}`
`2: {int} S = {1, 2, 3};`
`   int i = first(S);`
`3: tuple T{int x; int y;} {T} S = {<1,2>,<3,4>};`
MOSEL
`1: S = union(i in 1..4 | i<>2) {i*3}    // ergibt {3,9,12}`
`2: S := 1..3;`
`   i := getfirst(S)`
`3: keine Tupel-Sets möglich`

Tabelle 5: Syntaxbeispiele erweiterter Setfunktionalitäten

Eine weitere Funktionalität ist das Handling von geordneten Mengen (Ordered Sets). Sets weisen zwar durch die Reihenfolge der Angabe ihrer Elemente bereits eine implizite Ordnung auf, die beispielsweise für Indizierungen übernommen wird, jedoch kann in AMPL diese Ordnung durch das Attribut `ordered` (für zyklische Anordnung durch das Attribut `circular`) explizit festgeschrieben werden. In OPL und MOSEL ist dies nicht explizit erforderlich. AMPL und OPL verfügen über eine Reihe von Funktionen zum Zugriff auf geordnete Mengen, etwa `first` (erstes Element), `last` (letztes Element), `next` (nächstes Element), `prev` (vorheriges Element) etc. In MOSEL ist die Unterstützung für geordnete Mengen etwas begrenzter: Funktionen wie `getfirst` oder `getlast` sind nur auf Ranges an-

wendbar, nicht aber auf beliebige Sets. Die jeweiligen Zeilen 2 in Tabelle 5 zeigen Syntaxbeispiele für den Zugriff auf das erste Element einer geordneten Menge.
Neben einfachen Datentypen können Sets in AMPL und OPL auch aus zusammengesetzten Typen (Tupeln) bestehen, die für komplexere Indizierungsausdrücke verwendet werden können. Dabei müssen die Elemente des Tupels nicht notwendigerweise vom gleichen Typ sein, möglich wären beispielsweise aus einem Integer und einem String zusammengesetzte Tupel wie `(1,"a")`; gleichfalls sind auch n-Tupel möglich. OPL erfordert die Deklaration eines Tupel-Typs, was ähnlich der in anderen Programmiersprachen üblichen Deklaration von *Records* oder *Structs* ist.
Ein weiteres wichtiges Feature ist die Möglichkeit der Verwendung von Sets als Indizes über dünn besetzte Arrays. In den eingangs angegebenen Dokumentationen der Modellierungssprachen wird dafür häufig als Beispiel ein typisches Transportproblem aufgeführt, bei dem es gilt, die Kosten des Transports von Gütern zwischen Ausgangs- und Zielorten zu minimieren. Die AMPL-Anweisung `set S={{1,2},{"a","b"}}` ergibt das kartesische Produkt der beiden Mengen {1,2} und {"a","b"}, also (1, "a"), (1, "b"), (2, "a"), (2, "b"). Definiert man die Menge der Ausgangsorte beispielsweise als `set ORIG = {"O1", "O2", "O3"}` und die der Zielorte als `set DEST = {"D1", "D2", "D3")`, dann ergibt das kartesiche Produkt `set LINKS = {ORIG,DEST}` anschaulich die Matrix aller Kombinationen von Ausgangs- und Zielorten. Das so definierte 2-dimensionale Set LINKS kann zur Indizierung eines zweidimensionalen Arrays benutzt werden, das die Transportkosten zwischen zwei Orten enthält. Gibt es nur für einige Kombinationen aus Ausgangs- und Zielort eine Verbindung, so wäre die resultierende Transportkostenmatrix dünn besetzt (sparse). Um in effizienter Weise mit solchen dünn besetzten Arrays umzugehen, muss das Set LINKS als Teilmenge des kartesischen Produkts aus ORIG und DEST definiert werden, was in AMPL mit `set LINKS within {ORIG,DEST}` geschieht. Anschließend kann das dünn besetzte Array der Transprortkosten deklariert werden mit `param cost {LINKS}`. Die Daten enthalten dann nur noch die jeweiligen Transportkosten für tatsächlich existierende Index-Tupel aus ORIG und DEST. Die Verwendung von Iteratoren (in AMPL Dummy Indices genannt) mit dünn besetzten Arrays zeigt folgendes Beispiel einer AMPL-Restriktion (aus [Fourer03, S. 95]):

```
subject to Supply {i in ORIG}:
     sum {(i,j) in LINKS} Trans[i,j] = supply[i];
```

OPL verwendet für die Indizierung dünn besetzter Objekte ein lediglich syntaktisch unterschiedliches, aber inhaltlich gleiches Konzept. Auch MOSEL unterstützt Array Sparsity, allerdings mit einer von AMPL und OPL verschiedenen Benutzungsweise der Indexsets.

Arithmetische Ausdrücke

Arithmetische Ausdrücke entstehen aus der Verknüpfung von Konstanten oder Variablen in Skalar- oder Array-Form durch Operatoren und Funktionen. Die Modellierungssprachen verfügen über eine Vielzahl von Operatoren und Funktionen, wobei in Zielfunktion und Restriktionen nur solche Ausdrücke verwendet werden dürfen, die mit dem jeweiligen Modelltyp konform sind, also z.B. im Falle von MIP-Modellen nur lineare Ausdrücke. Die Grundoperatoren zur Addition, Subtraktion, Multiplikation, Division und Potenzierung (+, -, *, /, ^) sind in allen Sprachen vorhanden. Ebenso gibt es eine Reihe von numerischen Funktionen, die von allen Sprachen unterstützt werden, wie etwa Absolutwert, Kosinus, Exponentialwert, Sinus, Tangens, Wurzel und einige weitere verwandte Funktionen. Diverse Varianten der Funktionen `min` und `max` liefern die kleinsten bzw. größten Elemente einer Liste oder eines Arrays. Die Zeilen 1 in Tabelle 6 zeigen ein Beispiel für das Minimum einer Liste und die Zeilen 2 die für das eines Arrays. Interessanterweise können die `min`- und `max`-Funktionen in OPL auch in Restriktionen in Verbindung mit Entscheidungsvariablen eingesetzt werden: So wird beispielsweise der Ausdruck

```
Restriktion1:  min(i in 1..10) DecVar[i] >= 1;
```

von OPL automatisch und für den Benutzer transparent in eine CPLEX-konforme MIP-Formulierung umgesetzt.

AMPL
`1: param MyMin = min (1,2,3);`
`2: param MyMin = min {i in 1..10} a[i];`
`3: param MySum = sum {i in 1..10} a[i];`
`4: sum {i in 1..5, j in 1..10} coe[i,j] * DecVar[i,j];`
OPL
`1: int MyMin = minl(1,2,3);`
`2: int MyMin = min(i in 1..10) a[i];`
`3: int MySum = sum(i in 1..10) a[i];`
`4: sum(i in 1..5, j in 1..10) coe[i][j] * DecVar[i][j];`
MOSEL
`1: MyMin := minlist(1,2,3)`
`2: MyMin := min (i in 1..10) a(i)`
`3: MySum := sum (i in 1..10) a(i)`
`4: sum (i in 1..5, j in 1..10) coe(i,j) * DecVar(i,j)`

Tabelle 6: Syntaxbeispiele Summen und Min/Max

Die wohl wichtigste Funktion zur Formulierung arithmetischer Ausdrücke ist die Summierungsfunktion `sum`, die das Summierungszeichen ∑ der algebraischen Notation repräsentiert.

Fast alle mittleren und größeren Modelle beinhalten solche iterativen Summierungen. Die jeweiligen Zeilen 3 in Tabelle 6 zeigen ein Beispiel für eine einfache Summation über ein Array `a` von Integervariablen, deren Ergebnis in die Datenvariable `MySum` geschrieben wird. Die Zeilen 4 geben einen Ausdruck wider, der in Zielfunktion oder Restriktionen erscheinen kann und der die Verwendung der Summation über mehrere Indizes darstellt. Bis auf sehr kleine syntaktische Unterschiede arbeiten die drei Modellierungssprachen bei der Summation alle nach dem gleichen Prinzip. Dies gilt auch für die Möglichkeit der verschachtelten Summation und für den Gebrauch der Klammern, wobei die üblichen Vorrangregeln für Operatoren gelten (z.B. ist `sum {i in 1..10} a[i]+1` nicht gleich `sum{i in 1..10} (a[i]+1)`).

In Zusammenhang mit den arithmetischen Funktionen seien auch diverse Stringfunktionen erwähnt: Strings können in AMPL mit „`&`“ und in OPL und MOSEL mit „+“ zusammengefügt werden. Für die Extraktion von Substrings gibt es in AMPL und MOSEL den Befehl `substr` und in OPL die Methode `substring`. Die OPL String-Klasse besitzt darüber hinaus noch einige weitere spezifische Methoden, wie z.B. Groß- und Kleinbuchstabenwandlung oder Zugriff auf einzelne Zeichen.

Logische Ausdrücke

Logische Ausdrücke entstehen durch die Verwendung logischer Operatoren, und ihr Ergebnis ist ein boolescher Wert. Alle betrachteten Modellierungssprachen verfügen über die auch in Programmiersprachen gebräuchlichen Operatoren `and, or, not` sowie über Vergleichsoperatoren (`<, <=, =, <>, >, >=`). Zur Prüfung der Zugehörigkeit zu einer Menge gibt es die Operatoren `in` bzw. `not in`.

Die Operatoren `exists` und `forall` unterscheiden sich in AMPL, OPL und MOSEL: In AMPL sind `exists` und `forall` logische Operatoren zur Prüfung, ob mindestens ein Element (`exists`) bzw. alle Elemente (`forall`) einer Menge oder eines Arrays einer Bedingung entsprechen (z.B. `forall {i in 1..10} a[i]>10` ist wahr, wenn alle Elemente des Arrays >10 sind). In OPL ist ein dem AMPL `exists` entsprechender Operator nicht vorhanden, und `forall` ist kein Operator, sondern eine Schleifenanweisung. In MOSEL ist `forall` ebenfalls eine Schleifenanweisung, und `exists` prüft das Vorhandensein eines Elements in einem dynamischen oder einem dünn besetzten Array, nicht aber beliebige Bedingungen wie in AMPL.

Zur Ausführungssteuerung in Abhängigkeit von logischen Ausdrücken gibt es in den drei Modellierungssprachen `if-then-else`-Konstrukte, die sich lediglich syntaktisch leicht unterscheiden.

Variablen

Die Entscheidungsvariablen des Modells werden in den Modellierungssprachen ähnlich wie Datenvariablen angelegt und indiziert, wobei in der Regel syntaktisch kaum Unterschiede zwischen der Behandlung von Datenvariablen (in AMPL Parameter genannt) und Entscheidungsvariablen bestehen. Entscheidungsvariablen müssen als solche deklariert werden. Tabelle 7 zeigt in den Zeilen 1 ein einfaches Beispiel für die Deklaration einer kontinuierlichen Entscheidungsvariablen. Nach einem erfolgreichen Optimierungslauf werden die Entscheidungsvariablen mit den entsprechenden Lösungswerten gefüllt und können über geeignete Befehle (z.B. `display`) ausgegeben werden.

AMPL
`1: var MyVar;`
`2: var MyVar {j in MySet} >= mymin[j], <= mymax[j];`
OPL
`1: dvar float MyVar;`
`2: dvar float MyVar [i in MySet] in mymin[i] .. mymax[i];`
MOSEL
`1: MyVar: mpvar`
`2: MyVar: array(MySet) of mpvar  !Im Deklarationsblock`
`forall(i in MySet) setlb(MyVar(i), mymin(i))  !ausserhalb`
`forall(i in MySet) setub(MyVar(i), mymax(i))`

Tabelle 7: Syntaxbeispiele Entscheidungsvariablen

Die meisten Entscheidungsvariablen sind ein- oder mehrfach indiziert, wobei Ranges und Sets als Indizes dienen können. AMPL und OPL erlauben es, Ober- und Untergrenzen für Variablen bereits mit der Deklaration zu setzen, während MOSEL ein explizites Setzen dieser Bounds außerhalb des Deklarationsteils erfordert (s. Tabelle 7, Zeilen 2 als Beispiel).
Entscheidungsvariabeln können in linearen Ausdrücken der Form ∑*Konstante*Variable* verwendet werden.

Restriktionen

Lineare Restriktionen werden in AMPL, OPL und MOSEL in sehr ähnlicher Weise ausgedrückt. AMPL fasst alle Ausdrücke, die nicht explizit andere Bedeutungen haben (Zielfunktion, Deklaration etc.) als Restriktionen auf. Daher ist die Angabe des Schlüsselworts `subject to` optional und dient lediglich der besseren Lesbarkeit. OPL erfordert, dass alle

Restriktionen in einen `subject to {}`-Block eingeschlossen werden, und in MOSEL dürfen Restriktionen nur außerhalb des Deklarationsblocks erscheinen.
Restriktionen, die nur die Unter- oder Obergrenze einer Variablen darstellen (z.B. Zeilen 1 in Tabelle 8) werden entweder vom Preprozessor des Solvers oder von der Modellierungssprache selbst in Bounds umgesetzt. In AMPL beispielsweise kann über eine Konfigurationsvariable eingestellt werden, ob AMPL ein Preprocessing durchführen soll oder nicht. Ein solches Preprocessing betrifft nicht nur Bounds, sondern auch noch eine Reihe weiterer Optimierungen, die teilweise sogar Unlösbarkeiten erkennen können, bevor das Modell überhaupt an den Solver übergeben wird (vgl. [Fourer03, S. 275-282]).

AMPL
`1: subject to MyRestr: MyVar <= 100;`
`2: subject to MyRestr:` `  sum{i in 1..5} MyConst[i]*MyVar[i] <= 100;`
`3: subject to MyRestr {i in 1..5}: MyVar[i] <= 100;`
OPL
`1: MyRestr: MyVar <= 100;`
`2: MyRestr: sum(i in 1..5) MyConst[i]*MyVar[i] <= 100;`
`3: forall (i in 1..5) MyRestr: MyVar[i] <= 100;`
MOSEL
`1: MyRestr:= MyVar <= 100`
`2: MyRestr:= sum(i in 1..3) MyConst(i)*MyVar(i) <= 100`
`3: forall(i in 1..5) MyRestr:= MyVar(i) <= 100`

Tabelle 8: Syntaxbeispiele Restriktionen

Restriktionen können alle erlaubten linearen Operatoren enthalten, in der Praxis ist dies häufig die Summationsfunktion `sum()` (Zeilen 2, Tabelle 8). Ebenso häufig ist es, dass Restriktionen über eine oder mehrere Dimensionen indiziert sind, wie im Beispiel in den Zeilen 3 von Tabelle 8. Restriktionen müssen nicht notwendigerweise einen Namen tragen, aus Gründen der Lesbarkeit oder zur Referenzierung an anderen Stellen des Modells sind Restriktionen jedoch in den meisten Modellen benannt.
Alle Modellierungssprachen können Restriktionen mit den Vergleichsoperatoren <=, >=, = bilden, wobei die Modellierungssprachen die Restriktionen intern in kanonische Form bringen, unabhängig davon, wie sie der Benutzer angeordnet hat. Darüber hinaus gibt es noch sprachspezifische Operatoren: Hier seien vor allem logische Operatoren in OPL erwähnt, wie etwa `and, or, min` oder `max`, die Restriktionen wie z.B.

```
min(i in 1..5) X[i] <= 10
```

erlauben. In diesem Beispiel wird sichergestellt, dass das Minimum eines Arrays von Entscheidungsvariablen kleiner als zehn ist. Solche Formulierungen werden von OPL automatisch und für den Benutzer transparent in von CPLEX lösbare MIP-Formulierungen übertragen. AMPL und MOSEL bieten keine vergleichbare Funktionalität.

Schließlich seien noch die MOSEL-spezifischen Operatoren `is_sos1` und `is_partint` erwähnt, die in Form einer Restriktion Variablen als Special Ordered Sets oder Partial Integers deklarieren.

Zielfunktion

Zielfunktionen bestehen aus einem linearen Ausdruck und ggf. einem Namen. Die Namensvergabe für Zielfunktionen ist dabei in AMPL obligatorisch und in MOSEL optional, während OPL nur unbenannte Zielfunktionen kennt. Der Name dient nach Modellösung als Platzhalter für den Zielfunktionswert. AMPL und MOSEL erlauben die Positionierung der Zielfunktion an beliebigen Stellen innerhalb der Modellformulierung; OPL erfordert dagegen die Angabe der Zielfunktion vor den Restriktionen. Im Übrigen unterliegt die Formulierung von Zielfunktionen in den drei Modellierungssprachen lediglich kleineren syntaktischen Unterschieden.

AMPL
`1: minimize Cost:  sum {i in 1..5} MyConst[i] * MyVar[i];`
OPL
`1: minimize sum(i in 1..5) MyConst[i] * MyVar[i];`
MOSEL
`1: minimize(sum(i in 1..5) MyConst(i)*MyVar(i))`

Tabelle 9: Syntaxbeispiele Zielfunktionen

Stückweise Linearisierung

Mittels stückweiser Linearisierung lassen sich nicht-lineare Funktionen durch eine Abfolge linearer Funktionen approximieren. Eine stückweise Linearisierung mit n Abschnitten wird definiert durch einen Ausgangspunktpunkt *(x,y)*, n Steigungen und n-1 Stützpunkte.

Sprachfeatures zur stückweisen Linearisierung existieren in AMPL und OPL, nicht jedoch in MOSEL, wo eigene explizite Formulierungen, ggf. unter Verwendung von Special Ordered Sets, gewählt werden müssen. Ausdrücke zur stückweisen Linearisierung können sowohl in der Zielfunktion, als auch in Restriktionen erscheinen. Die Beispiele in Tabelle 10 zeigen stückweise Linearisierungen einer Quadratfunktion für die Intervalle [0..2], [2..4], [4..6], [6..∞) in Zielfunktion und Restriktionen. In AMPL werden innerhalb der spitzen Doppelklammer << >> zuerst die Stützpunkte und dann die Steigungen angegeben. OPL erwartet innerhalb des `piecewise{}`-Blocks die Angaben einer Liste mit Elementen der Form `Stützpunkt->Steigung`. In OPL kann der Ausgangspunkt frei gewählt werden, während AMPL immer vom Ausgangspunkt (0,0) ausgeht.

AMPL
`minimize Cost: << 2,4,6; 2,6,10,20 >> X;`
`subject to Restr1: X2 = << 2,4,6; 2,6,10,20 >> X;`
OPL
`minimize ( piecewise{2->2; 6->4; 10->6; 20}(0,0) X);`
`Restr1: X2 == piecewise{2->2; 6->4; 10->6; 20}(0,0) X;`

Tabelle 10: Syntaxbeispiele Stückweise Linearisierung

Stückweise Linearisierungen werden von der Modellierungssprache, für den Benutzer transparent, in entsprechende LP- oder MIP-Formulierungen übertragen, die von einem geeigneten Solver gelöst werden können. Dabei wird ein konvexer, separabler Kurvenverlauf in der Zielfunktion bei Minimierung und ein konkaver, separabler Verlauf bei Maximierung in eine reine LP-Formulierung übertragen. Gleiches gilt für konvexe Kurvenverläufe in ≤ Restriktionen und konkave Verläufe in ≥ Restriktionen. In alle anderen Fällen wird eine MIP-Formulierung mit Binärvariablen erzeugt.

Prozedurale Sprachelemente (Scripts)

AMPL, OPL und MOSEL besitzen einen gemeinsamen Kern an prozeduralen Sprachelementen, der weitgehend identisch ist mit auch in anderen Programmiersprachen vorhandenen Strukturen. In OPL werden diese prozeduralen Elemente als *Scripts* bezeichnet.

AMPL
`1: for {i in 1..4} {…};`
`2: repeat while x > 0 {…};`
`3: repeat {…} until x > 0;`
`4: if MyParam < 10 then {…} else {…};`
OPL
`1: for (var i=1; i <= 4; i++) {…};`
`2: while (x > 0) {…};`
`3: repeat {…} until x > 0;    // Seit OPL 4.x entfallen`
`4: if (MyParam < 10) {…} else {…};`
MOSEL
`1: forall (i in 1..4) do … end-do`
`2: while(x > 0) do … end-do`
`3: repeat … until x > 0`
`4: if MyParam < 10 then … else … end-if`

Tabelle 11: Syntaxbeispiele Schleifen und Bedingungen

Abgesehen von kleineren syntaktischen Unterschieden existieren in den Modellierungssprachen die gleichen, typischen Schleifenstrukturen `for…`, `while…do` und `repeat…until`. AMPL und OPL haben die gleichen Befehle für Abbruch und Fortsetzung einer Schleife: `break` und `continue`. MOSEL besitzt dazu die ähnlichen Befehle `break` und `next`. Ebenso existieren überall `If…Then`-Konstrukte Zur Ausgabe wird in AMPL `print`, `printf` oder `display` und in OPL und MOSEL `write` oder `writeln` verwendet. Das Einlesen von Daten ist in AMPL aus unformatierten Dateien mit dem Befehl `read` möglich,

OPL liest aus Spreadsheets und Datenbanken ein (`DBread`), und MOSEL kann mit `read` und `readln` vom jeweiligen Input-Stream lesen.
Über diese grundlegenden Sprachelemente hinaus existiert noch eine Reihe weiterer Features, die in den drei Modellierungssprachen sehr ähnlich sind, wie etwa benutzerdefinierte Funktionen und Prozeduren, die in OPL innerhalb eines `function{}`-Blocks und in MOSEL in `procedure … end-procedure` bzw. `function … end-function` Blocks definiert werden. In AMPL besteht lediglich die Möglichkeit, benutzerdefinierte Funktionen aus eigenen externen Bibliotheken (DLLs) zu benutzen.

Daten und Datenbankanbindung

Alle drei Modellierungssprachen trennen Modellstruktur und Modelldaten voneinander. AMPL und OPL gebrauchen hierzu Dateien mit Endung `.mod` für das Modell und `.dat` für Datenfiles. MOSEL benutzt die Endungen `.mos` und `.dat`. Für Modelle mit größeren Datenvolumen ist häufig die Anbindung an relationale Datenbanken erforderlich. AMPL hat hierfür eine spezielle Handle-Bibliothek (ampltabl.dll), die Verbindungen via ODBC zu Datenbanken oder zu Excel ermöglicht. Mittels entsprechender Befehle (`table, read table, write table` etc.) kann AMPL nicht nur Daten aus ODBC-Quellen lesen, sondern auch Ergebnisse dorthin zurückschreiben und SQL-Befehle senden. OPL verfügt ebenfalls über eine derartige bidirektionale ODBC-Anbindung, bietet darüber hinaus aber auch die Möglichkeit der direkten Kommunikation mit diversen Datenbanken und mit Excel. Ähnliches leistet auch MOSEL, wobei hier die Datenbankfunktionalitäten entsprechend der MOSEL-typischen Modularchitektur in einem eigenen Modul (*mmodbc*) vereinigt sind.

3.4.2 Besonderheiten der untersuchten Modellierungssprachen

Nachdem im vorherigen Abschnitt auf die wesentlichen gemeinsamen Sprach- und Modellierungselemente eingegangen wurde, sollen nun die Unterschiede der drei betrachteten Modellierungssprachen gegenübergestellt werden. Dabei geht es vor allem um architektonische und strukturelle Unterschiede – Details wie einzelne unterschiedliche Funktionen sollen außer Betracht bleiben.

Architektur

Die verfügbaren AMPL-Versionen für Windows und Unix sind kommandozeilenorientierte ausführbare Dateien. Unter Windows beinhaltet die Datei *ampl.exe* neben der Benutzerschnittstelle alle für die Übersetzung von AMPL-Modellen notwendigen Routinen und Funk-

tionalitäten. Diese Kompaktheit macht die Verteilung und Installation von AMPL sehr einfach. Das Kommandozeileninterface hat jedoch den Nachteil, dass sich größere Modelle nur mühsam bearbeiten lassen, weshalb der Einsatz von grafischen Benutzeroberflächen von Drittherstellern oder das in dieser Arbeit entwickelte MOPS Studio fast unverzichtbar sind. Der Austausch von Modelldaten und Optimierungsergebnissen mit angebundenen Solvern erfolgt über Interimsdateien (*Stubs*).
Auch OPL besitzt eine Kommandozeilenschnittstelle in Versionen für Windows (*oplrun.exe*) und UNIX. Im Gegensatz zu AMPL ist dies jedoch nur eine optional benutzbare zusätzliche Schnittstelle; die eigentlichen Funktionalitäten befinden sich unter Windows in einer Reihe von DLLs, die von der eng an die Sprache gekoppelten Umgebung *OPL-Studio* angesprochen werden. Ebenfalls als DLL eingebunden wird CPLEX, der von OPL exklusiv genutzte Solver. Diese Architektur ermöglicht schnellen Datenaustausch im Hauptspeicher, ohne dass – wie bei AMPL – Interimsfiles notwendig sind.
Die Architektur von MOSEL unterscheidet sich durch ein erweiterbares Modulkonzept und die Integrationsmöglichkeiten neuer Sprachelemente. Solver und andere Funktionseinheiten, wie Datenbankkonnektoren (z.B. *mmodbc*) und Grafikbibliotheken (z.B. *mmive*) werden als Module in MOSEL eingebunden und können eigene Befehle und Funktionsaufrufe definieren, die wie originäre MOSEL-Befehle aufgerufen werden. So definiert das Solver-Modul *mmxprs*, das die XPRESS-Optimierungsengine einbindet, beispielsweise die Befehle `maximize` und `minimize` zum Start des Optimierungslaufs. Module kommunizieren dabei mit dem Kernsystem über das *MOSEL Native Interface*, das sich aus C-Programmen ansprechen lässt. Zwar besitzt auch MOSEL eine Kommandozeilenschnittstelle (*mosel.exe*), ähnlich wie bei OPL ist diese jedoch nur eine Kapsel, während sich der Großteil der funktionalen Routinen in mehreren DLLs befindet.

Solveranbindung

Weiterhin unterscheiden sich die Systeme durch unterschiedliche Möglichkeiten der Solvereinbindung, was vor allem Konsequenzen für die Performance des Lösungsprozesses und dessen Steuerung hat. AMPL ist von seinen Interfaces völlig offen konstruiert, so dass mit einem überschaubaren Programmieraufwand (vgl. [Gay97]) fast jeder Solver anbindbar ist. Anfang 2007 listet die AMPL Website ([AMPL07a]) 35 verschiedene anbindbare Solver für lineare und nicht-lineare Modelle auf (einschließlich Komplementär- und Netzwerkmodelle).
Anders im Falle von OPL, das als ILOG-Produkt fest mit dem ILOG-Optimierer CPLEX verbunden ist. Frühere Versionen von OPL unterstützen auch Constraint Programming mit einer

Anbindung an den ILOG CP-Solver. Wie [vHentenr99] beschreibt, bot diese enge Verzahnung von Solver und Modellierungssprache vielfältige Möglichkeiten, Suchalgorithmen für das Constraint Programming in der Modellierungssprache zu definieren. In der aktuellen OPL Version 5.0 von 2006 ist diese Anbindung für den CP-Solver jedoch weggefallen, und es wird nur noch CPLEX unterstützt.

MOSEL erlaubt dank seines modularen Konzepts zwar prinzipiell die Verwendung unterschiedlicher Optimierungsengines, aufgrund der Produktpolitik des Herstellers (Auslieferung als komplettes System) kommt es bei MOSEL faktisch jedoch zu einer engen Koppelung an die Optimierungsengine XPRESS. Obwohl sich diese Arbeit auf lineare Modelle fokussiert, sei erwähnt, dass die drei Modellierungssprachen auch nicht-lineare Modellierungen unterstützen. Bei AMPL hängen die Möglichkeiten nicht-linearer Programmierung primär vom eingesetzten Solver ab, OPL bietet quadratische Programmierung im Rahmen von CPLEX, und MOSEL erlaubt nicht-lineare Modellierungen bei Vorhandensein entsprechender Module (z.B. *mmquad* für quadratische Programmierung).

Die unterschiedlichen Einbindungsmöglichkeiten von Solvern haben weitergehende Auswirkungen auf Sprachumfang und -struktur. So existieren in AMPL einige Schlüsselwörter, die nur in Verbindung mit bestimmten Solvern anwendbar sind, wie beispielsweise `complements` zur Formulierung von Komplementärproblemen (vgl. [Fourer03, S. 419-435]), die z.B. mit dem Solver *Path* ([Path07]) gelöst werden können. Der weitaus größte Teil des AMPL-Sprachumfangs ist jedoch mit allen Solvern nutzbar. Interessant ist dabei, dass sich etwa Netzwerkmodelle mit speziellen Modellierungsobjekten wie `node` und `arc` formulieren lassen und sowohl von Solvern gelöst werden können, die spezielle Netzwerkoptimierungsalgorithmen besitzen (wie z.B. CPLEX) als auch von solchen, die nur allgemeine LP/MIP-Modelle unterstützen (wie z.B. MOPS). Die Umsetzung erfolgt durch die jeweiligen Solver-Interfaces und ist für den Benutzer weitestgehend transparent.

Quellcode-Verarbeitung

Ein weiterer Unterschied liegt in der Art und Weise, wie die Engines der Modellierungssprachen den Sourcecode des Modells verarbeiten. AMPL erzeugt aus dem Quelltext des Benutzers und den Datenfiles ein so genanntes Stub-File, das von der AMPL-Solverschnittstelle für die jeweils verwendete Optimierungsengine weiterverarbeitet wird. Allerdings sind diese Stub-Files normalerweise für den Benutzer nicht relevant, so dass sich bei AMPL aus Benutzersicht ein einstufiger Prozess ergibt, der Ähnlichkeit mit der Sourcecode-Verarbeitung von Interpreter-Programmiersprachen hat. Anders MOSEL, das einen zweistufigen Verarbei-

tungsprozess verwendet, der dem einer Compiler-Programmiersprache ähnelt. Ausgehend von der textualen Repräsentation des Modells erzeugt MOSEL (genauer gesagt die MOSEL Compiler Library) zunächst ein kompiliertes Modell in binärem Format (BIM, Binary Model), das in einem zweiten Schritt von der MOSEL Runtime Library ausgeführt wird. Die Runtime Library enthält die MOSEL Virtual Machine, die unter anderem auch für das Management der MOSEL-Module zur Laufzeit zuständig ist. Zur Einbindung in eine Applikation wird das kompilierte Binary Model verwendet, was zur Folge hat, dass das geistige Eigentum am Modell bei Weitergabe der Applikation an Dritte geschützt bleibt. Die Ausgangsdaten in den Datenfiles sind davon nicht betroffen und können auch nach Kompilation beliebig verändert werden. Weiterhin muss bei Weitergabe der Applikation lediglich die MOSEL Runtime Library zur Verarbeitung der Binary Models eingeschlossen werden. Auch OPL bietet optional die Möglichkeit, Modelle zu kompilieren und in Binärform weiterzugeben, jedoch ohne dass dies zwingend erforderlich ist.

Prozedurale und deklarative Elemente

Alle drei Modellierungssprachen enthalten sowohl deklarative Elemente, die die eigentliche Modellbeschreibung enthalten, als auch prozedurale Elemente, die Algorithmen für Datenaufbereitung, Lösungsprozesse oder Ein- und Ausgabe definieren. Die fehlende Notwendigkeit der Kompilation und größere Freiheiten bei der Abfolge der modelldefinitorischen Befehle geben AMPL im Vergleich zu den anderen beiden Sprachen einen stärker deklarativen Charakter. Zugleich enthält AMPL jedoch auch viele prozedurale Sprachelemente, wie etwa Schleifen (`for`, `repeat until`, `repeat while`), Prüfbedingungen (`if-then-else`), Stringbearbeitung (`substr`, `length` etc.) oder Ausgabeanweisungen (`display`). Die Offenheit des Solverinterfaces in AMPL setzt einer prozeduralen Interaktion mit Solvern während des Lösungsprozesses engere Grenzen als etwa bei OPL, das nur mit einem einzigen Solver interagiert.

OPL verfolgt den Grundsatz einer klaren Trennung von deklarativen und prozeduralen Elementen. Die prozeduralen Parts werden als *OPL Script* bzw. neuerdings als *ILOG Script* bezeichnet und sind auch syntaktisch durch die Abgrenzung in `execute{}`-Blocks von den modellbeschreibenden Parts getrennt. [vHentenr00] beschreibt den Einsatz von OPL Script beispielsweise zur Lösung einer Abfolge von Modellen oder zur Formulierung von Column Generation-Problemen (was freilich auch mit anderen Modellierungssprachen möglich ist).

In MOSEL können prozedurale und deklarative Elemente beliebig gemischt werden – egal, ob es native oder in einem Modul definierte Sprachelemente sind. Der Grundcharakter von MO-

SEL ist dabei jedoch prozedural: Alle Sprachelemente werden in der Reihenfolge ihres Auftretens im Code kompiliert und ausgeführt. Hierin unterscheidet sich MOSEL von OPL, das eine spezifische Processing-Reihenfolge einhält (zuerst Datenquellen, dann `execute`-Blocks, dann `assert`-Befehle).

Special Ordered Sets

Das Konzept der Special Ordered Sets (SOS) wurde zuerst von Beale, Tomlin und Forrest vorgeschlagen (vgl. [Beale76], [Beale70]) und wird mittlerweile von den meisten MIP-Solvern unterstützt. Man unterscheidet die Typen 1,2 und 3, wobei die gebräuchlichsten Typ 1 und Typ 2 sind: Ein SOS vom Typ 1 ist eine geordnete Menge von Entscheidungsvariablen, von denen maximal eine ungleich Null sein darf. In einem SOS vom Typ 2 dürfen lediglich maximal zwei benachbarte Entscheidungsvariablen ungleich Null sein.
Von den hier betrachteten Modellierungssprachen bietet lediglich MOSEL die Möglichkeit, SOS explizit zu deklarieren, wie das folgende Syntaxbeispiel für die Deklaration eines aus zehn Variablen bestehenden Special Ordered Sets Typ 1 zeigt:

```
sum(MyWeight in 1..10) MyWeight * MyVar(i) is_sos1
```

Der Laufindex `MyWeight` dient dabei der Gewichtung der Variablen und erzeugt so mit aufsteigender Gewichtung die erforderliche Reihenfolge.
OPL und AMPL besitzen keine expliziten Sprachfeatures für Special Ordered Sets. In OPL kann als Workaround ggf. nach Übergabe des Modells an CPLEX mittels CPLEX-Optionen und -Befehlen ein SOS nachträglich definiert werden. AMPL besitzt gleichfalls keine SOS-Informationen, die an einen Solver übergeben werden könnten. Das Vorhandensein einer Option zur automatischen Identifikation von SOS durch den Solver (z.B. CPLEX-Option `sosscan` zur Identifikation von SOS3) bringt lt. AMPL FAQs ([AMPL07b]) ebenfalls meist keine wesentliche Beschleunigung des Branch-and-Bound-Prozesses.

Debugging

Die Debugging-Features der Modellierungssprachen beziehen sich fast ausschließlich auf die prozeduralen Sprachbestandteile, für die das Debugging ähnlich wie in anderen Programmiersprachen und -umgebungen funktioniert. Ein echtes Debugging des Modells in Bezug auf Unlösbarkeiten und deren Analyse ist nicht möglich.
Da AMPL lediglich eine Kommandozeilenschnittstelle besitzt, muss dort das Debugging ebenfalls kommandozeilenbasiert erfolgen. Über die AMPL Option `single_step` lässt sich

die Schrittweite für das Debugging festlegen und mit den Befehlen `step` und `next` können die jeweils nächsten Schritte oder Blöcke des Models zur Ausführung gebracht werden.
Komfortabler sind die Debuggingmöglichkeiten von OPL und MOSEL aus den jeweiligen integrieren Entwicklungsumgebungen. Dort lassen sich Breakpoints setzen, und die Werte von Entscheidungs- und Datenvariablen können über die integrierten Debugging-Explorer angezeigt und analysiert werden. MOSEL erlaubt dabei im Gegensatz zu OPL auch das Setzen von Breakpoints innerhalb von Deklarationen und nicht nur in den prozeduralen Parts. MOSEL bietet weiterhin einen Commandline-Debugger, dessen Funktionsumfang größer als der des AMPL-Debuggers ist.

Einbettungsmöglichkeiten in Applikationen

Die Modellierungssprachen unterscheiden sich stark hinsichtlich der Möglichkeit, Modelle in benutzererstellte Anwendungen einzubetten: AMPL ist von seinem Grundkonzept auf kommandozeilenbasierte Batch-Verarbeitung ausgerichtet, und demzufolge ist die Applikationseinbettung relativ schwierig. Die im Rahmen dieser Arbeit erstellten Komponenten sind ein Versuch, diesen Mangel zu beheben. Ein ähnlicher Ansatz in diese Richtung ist die *AMPL COM Library* von OptiRisk Systems ([OptiRisk07]), die allerdings zum Erstellungszeitpunkt (ca. 2005) der hier implementierten Bibliotheken noch nicht auf dem Markt war.
Ganz anders im Falle von OPL und MOSEL, die umfangreiche Bibliotheken zur Anwendungseinbettung besitzen. OPL kann sowohl unter Unix/Linux in C++-Programme, als auch unter Windows in C++, Java und .NET-Anwendungen eingebettet werden. Dabei existiert eine enge Verbindung der OPL-Bibliothek zu den CPLEX- und Concert-Bibliotheken. So können über die CPLEX API beispielsweise Parameter oder Callbacks genutzt werden, und über Concert-Klassen kann auf Bestandteile des Modells zugegriffen werden. Das OPL-Modell kann als Plaintext oder in kompilierter Form zur Laufzeit der Anwendung von entsprechenden Objekten der OPL-Bibliothek eingelesen werden. Auch bei der Applikationseinbindung herrscht eine strikte Trennung von Modell und Daten. So kann das gleiche Modell mit unterschiedlichen Daten mehrfach instanziert werden. Ebenso sind Modellierungsphase und Lösungsphase voneinander getrennt.
Die MOSEL-Einbettungsbibliotheken bestehen aus zwei Teilen: Die *MOSEL Model Compiler Library* dient lediglich dazu, ein Modell von Sourcecode in Binärcode (BIM) zu übersetzen. Die *MOSEL Run Time Library* ist der eigentlich relevante Teil um kompilierte MOSEL-Modelle in einer Anwendung einzulesen und zu lösen. Normalerweise wird nur die Run Time Library in einer Applikation genutzt, und mit dieser ausgeliefert. Über entsprechend definierte

Typen (`XPRMmpvar`, `XPRMlinctr`, `XPRMset`, `XPRMarray` etc.) kann der Zugriff auf alle Modellbestandteile wie Variablen, Restriktionen, Sets, Arrays etc. sowie auf die Lösung erfolgen.

3.4.3 Vergleich ausgewählter Eigenschaften

In Abschnitt 3.4 wurden in Anlehnung an [Schichl04] und [Kallrath04b] eine Reihe idealtypischer Features moderner Modellierungssprachen und -systeme aufgelistet. Die hier betrachteten Sprachen werden in der folgenden Tabelle 12 anhand dieser Kriterien gegenübergestellt.

Kriterium	AMPL	OPL	MOSEL
Syntaktische Nähe zur algebraischen Notation	hoch	hoch	hoch
Sprachliche Abdeckung aller Features der potenziell benutzbaren Solver	gering	mittel	modulabhängig
Anbindung möglichst vielfältiger Solver	hoch	gering	hoch
Offenheit bzgl. Datenanbindung	mittel	hoch	hoch
Unabhängigkeit von Modellstruktur und Modelldaten	ja	ja	ja
Plattformunabhängigkeit	hoch	mittel	mittel
Skalierbarkeit bzgl. Performance und Ressourcenverbrauch[1)]	ja	ja	ja
Einbettungsmöglichkeiten in Anwendungen	gering	hoch	hoch
Unterstützung Infeasibility-Analyse	ja	ja	ja[2)]
Effizientes, flexibles Indexing	ja	ja	ja
Versionsmanagement (Verwaltung mehrer Modellinstanzen)[3)]	ja	ja	ja
Verteilte Entwicklung und Optimierung[4)]	nein	nein	nein
Möglichkeit der Steuerung des Lösungsprozesses[5)]	gering	hoch	modulabhängig

[1)] Wurde nicht getestet. Wesentlich Performance- und Ressourcenprobleme sind aber nicht bekannt.
[2)] Zumindest bei Verwendung des XPRESS Solver Moduls *mmxprs* (via Funktion *getiis*)
[3)] Andere Arten von Versionsmanagement sind nicht in den Sprachen selbst, sondern nur in den integrierten Entwicklungsumgebungen möglich.
[4)] Wird z.B. über Infrastrukturen wie den Neos-Server ([Dolan02], [NEOS06]) ermöglicht, nicht aber innerhalb der Sprachen selbst.
[5)] Beispielsweise Callbacks oder Definitionsmöglichkeiten für eigene Lösungsalgorithmen.

Tabelle 12: Vergleich Features Modellierungssprachen