ES-FLEX-INFRA Dokumentation

Release 1.0

ES-FLEX-INFRA Team

31.12.2019

Inhaltsverzeichnis

KAPITEL 1

Algorithmik

Dieses Dokument fasst die algorithmischen Aspekte des Projekts zusammen, die seitens der TH Köln bearbeitet worden sind.

1.1 Mathematische Optimierung

Die mathematische Optimierung bestimmt im Allgemeinen die Menge der Entscheidungsvariablen eines Systems unter Einbeziehung einer Zielfunktion und unter der Einhaltung von Nebenbedingungen (Constraints). Ohne weitere Einschränkungen liegen die Beschreibungen (in der Praxis) in der komplexesten Problemklasse **Mixed-Integer Nonlinear Programming (MINLP)** vor:

$$\begin{aligned} &\min_{X,Y} \ f(X,Y) \\ &s.t. \ g(X,Y) \leq 0 \end{aligned}$$

Dabei sind $f : \mathbb{R}^n \to \mathbb{R}$ und $g : \mathbb{R}^n \to \mathbb{R}^m$ lineare oder nichtlineare Funktionen und $X \subseteq \mathbb{R}^{n_1}$, $Y \subseteq \mathbb{Z}^{n_2} (n := n_1 + n_2)$ die kontinuierlichen und diskreten Entscheidungsvariablen.

Aufgrund der enthaltenen ganzzahligen Anteile ist das Problem **NP-schwer**, gehört also zu den hartnäckigen Problemstellungen. Im Kapitel Software wird die projektbezogene Wahl eines entsprechenden Lösers (Solvers) diskutiert.

Innerhalb dieses Projekts umfasst die mathematische Optimierung im Wesentlichen die folgenden, nach Wichtigkeit priorisierten, Aspekte:

(a) Bestimmen der Flussvariablen für alle Zeitschritte.

(b) Bestimmen der kontinuierlichen *Parameter* von Speicher- und Konvertertechnologien, wie z.B. die Dimensionierung; in Abhängigkeit von den anfallenden Kosten.

(c) Bestimmen der *Typen* der Technologien, die in das Netz zu integrieren sind.

(d) Bestimmen der geometrischen *Standorte* (Lokationen).

Ziel ist es, ein optimales Mengengerüst zu finden, dass die insgesamt anfallenden Treibhausgase minimiert. Dabei wird jeweils von einer endlichen zur Verfügung stehenden monetären Investitionssumme ausgegangen. Diese wird als Nebenbedingung formuliert.

Dieses Kapitel stellt eine Reihe von Modellen vor, die auf die Lösung des Gesamtproblems hinarbeiten. Der Detaillierungsgrad wird sukzessive erhöht; die Berechnungskomplexität nimmt jeweils zu, und reicht von der *Linearen Programmierung* bis hin zur *Gemischt-ganzzahligen Nichtlinearen Programmierung*. Einige Fragestellungen aus dem Forschungsprojekt können bereits mit einfachen Modellen gelöst werden. Aufgrund der immensen Instanzgrößen aus der Praxis und der teilweise enthaltenen Ganzzahlanteile müssen die Formulierungen mit teils starken Relaxionen versehen werden. Insbesondere die beschriebenen physikalischen Zusammenhänge gilt es zu vereinfachen, um ein *lösbares* Problem zu erhalten.

Im Kapitel Software findet man eine implementierende Softwarekomponente „Optimizer“. Die Formulierung der Optimierungsaufgabe erfordert es, die in sich geschlossene Gesamtheit an Nebenbedingungen zu formulieren. Nur als Blackbox definierte Konstrukte (z.B. Teilsimulationen) können nicht einbezogen werden, da es in diesem Falle z.B. nicht möglich wäre, die partiellen Ableitungen der Zielfunktion und der Nebenbedingungen zu definieren.

Der gewählte Gesamtoptimierungsansatz sieht vor, dass man

(a) eine heuristische Optimierung vornimmt, um Kenngrößen abzuschätzen (Komponente „Optimizer“) *und*

(b) per *physikalischer Simulation* die Validität der Postulate von a) verifiziert (Komponenten „MYNTS“, „PyPSA“ usw.)

Die Modellierung des Optimierungsproblems soll im Folgenden in der Formalisierung so homogen wie möglich dargestellt werden. Ziel ist, dass die Menge der erzeugten Modelle systematisch durch eine automatisierte, softwaregesteuerte Erzeugung von mathematischen Ausdrücken abgebildet werden kann. Bei einfachen Problemstellungen wird so ein Großteil der Entscheidungsvariablen durch Konstanten ersetzt. Die Laufzeit kann entsprechend gering gehalten werden.

1.1.1 Notation

In diesem Kapitel wird, sofern nicht explizit erwähnt, eine graphentheoretische Notation verwendet, auch wenn die Bezeichner im Bereich der erneuerbaren Energien teilweise anders lauten. Zum Beispiel wird der Begriff *Kapazität* als maximaler Durchfluss für eine Leitung des Flussproblems verstanden. Es wird darauf geachtet, stets Klarheit zu wahren. Als abkürzende Notation für die Menge $\{1, 2, \ldots, n\}$ schreiben wir $[n]$. Zeitreihen der Form $\{x_1, x_2, \ldots, x_T\}$ werden verkürzt durch $\{x_t\}_{t=1}^{T}$ ausgedrückt.

1.1.2 Modellierung

Wie bereits erläutert, müssen im Rahmen der Modellierung des Gesamtoptimierungsproblems aufgrund der NP-schwere teilweise drastische physikalische Relaxierungen vorgenommen werden.

Als Zielfunktion wird stets die Minimierung der systemweit anfallenden Treibhausgase angenommen. Betriebswirtschaftliche Aspekte, wie die Summe der anfallenden Investitionskosten für neue Technologien und laufende Energiekosten, werden als Nebenbedingungen formuliert. Während die physikalischen Gleichungen erhebliche Nichtlinearitäten vorgeben, wird hier wo möglich, eine Linearisierung vorgenommen.

Sei n die Anzahl der Entscheidungsvariablen und sei m die Anzahl der Nebenbedingungen. Dann definieren wir das Optimierungsproblem in standardisierter Form wie folgt:

$$\begin{aligned} \min_{x} \quad & f_0 \\ & f_i < 0 \qquad \forall i \in [m] \end{aligned}$$

Dabei sei $x \subseteq \mathbb{R}^n$, sowie $x_i \in \mathbb{Z}$ für $i \subseteq [n_1]$ $(n_1 < n)$.

Die Optimierung wird als Programm über T Zeitschritte formuliert, also $t \in [T]$. Die Zeitdifferenz zwischen zwei aufeinanderfolgenden Zeitschritten beträgt i.d.R. 15 Minuten.

Wo möglich wird Energie, unabängig vom Energiesektor, immer in der Einheit Kilowattstunden (kWh) ausgedrückt. Entsprechend entfällt der Konvertierungsaufwand und Inkonsistenzen werden vorgebeugt.

Netzwerk / Graph

Für die Gesamtoptimierung wird das multisektorale Energienetz über einen (gerichteten) Graphen $G = (V, E)$ dargestellt. Die Energiesektoren werden durch eine Knotenpartitionierung abgebildet: Wir notieren für die Subgraphen $G_i \subseteq G$. Der Energiesektor selbst wird durch den Intex i gelabelt.

Knoten

Die Knotenmenge $V(G)$ ist wie folgt partitioniert:

- Energieerzeuger (Quellen) A.
- Energieverbraucher (Senken) B.
- Speicherknoten S.
- Sonstige Knoten N.

Jeder Knoten enthält mindestens die folgenden Attribute:

- $p(v)$ geometrische Position in Form von Breiten- und Längengraden (vornehmlich nach WGS84).
- $p_t^+(a)$ abrufbare Energie am Erzeugerknoten $a \in A$ zum Zeitpunkt $t \in [T]$.
- $p_t^-(b)$ abgefragte/verbrauchte Energie am Verbraucherknoten $b \in B$ zum Zeitpunkt $t \in [T]$.
- $\gamma(a)$ anfallende Treibhausgase durch Energieerzeugung am Knoten $a \in A$ pro abgerufender Kilowattstunde in g/kWh.

Kanten

Die Kantenmenge $E(G)$ ist wie folgt partitioniert:

- Energiekonverter C
- Mit Energiespeichern inzidente Kanten S
- Zwischenkanten N

Eine Kante $e \in E(G)$ bildet je eine physikalische Leitung des Energienetzes ab. Kanten enthalten die folgenden Attribute:

- $c(e)$ Leitungskapazität in Form der maximal pro Zeiteinheit über die Leitung transportierbarer Energie in Kilowattstunden.
- $\eta(e)$ Wirkungsgrad der Leitung $e \in E$ zur linear approximativen Abbildung von Übertragungsverlusten.
- $f_t(e)$ beschreibt den Energiefluss zum Zeitpunkt t auf der Leitung $e \in E$.

Speichertechnologien

In der gröbsten Approximationsstufe werden Speichertechnologien $s \in S \subseteq V(G)$ wie folgt definiert:

- $c(s)$ maximale Ladung / Kapazität in Kilowattstunden. Im Falle der Bestimmung durch Optimierung ist $c(v)$ eine Entscheidungsvariable, die durch eine untere und obere Grenze beschränkt wird.
- $\eta_i(s) \in [0, 1]$ Wirkungsgrad des Einspeicherns.
- $\eta_o(s) \in [0, 1]$ Wirkungsgrad des Ausspeicherns.
- $\hat{k}(s) = \hat{k}_1 \cdot c(s) + \hat{k}_0$ Investitionskosten in Abhändigkeit der Speicherkapazität.
- $\tilde{k}(s) = \tilde{k}_1 \cdot f_t(u, s)$ Optional: Laufende Kosten in Abhängigkeit des Energieflusses in $[EUR]$.
- $\gamma(s)$ Durch die Technologie im laufenden Betrieb erzeugte Treibhausgase in [g/kWh].

Konvertertechnologien

In der größtmöglichen Approximationsstufe werden Konvertertechnologien $c \in C \subseteq E(G)$ wie folgt definiert:

- $c(c)$ maximaler Durchfluss in Kilowattstunden. Im Falle der Bestimmung durch Optimierung ist $c(c)$ eine Entscheidungsvariable, die durch eine untere und obere Grenze beschränkt wird.
- $\eta(c) \in [0, 1]$ Wirkungsgrad der Konvertierung.
- $\hat{k}(c) = \hat{k}_1 \cdot c(c) + \hat{k}_0$ Investitionskosten in Abhängigkeit von dem maximalen Durchfluss.
- $\tilde{k}(c) = \tilde{k}_1 \cdot f_t(c)$ Laufende Kosten in Abhängigkeit des Energieflusses.
- $\gamma(c)$ Durch die Technologie im laufenden Betrieb erzeugte Treibhausgase in [g/kWh].

Komplexere Konvertertechnologien werden als *Symbiose* der oben definierten Primitive (Speicher und einfache Konverter) implementiert.

Im Folgenden wird eine Menge von zunächst theoretischen Modellen (nach aufsteigendem Schweregrad bei der Bestimmung der Lösung) aufgelistet. Die mathematische Problemformulierung wird dabei sukzessive ausgebaut. Ein Modell $i + 1$ erbt alle Nebenbedingungen von Modell i, sofern dies nicht explizit im Text ausgeschlossen wird.

Modell 1: Flußproblem ohne Technologien

Die Standorte für Technologien sind bekannt, sowie alle Parameter vordefiniert. Die Ansteuerung der Technologien (Ein- und Ausspeichern bei Speichern, sowie Durchsatz bei Konvertern) erfolgt über vordefinierte Zeitreihen. Bei der Lösung des unterliegenden Flussproblems wird davon ausgegangen, dass zunächst stets die zum Zeitpunkt $t \in [T]$ verfügbaren erneuerbaren Energien von Erzeugern mit den geringsten Treibhausgasen verwertet werden. Diese Betrachtungsweise ist legitim, da im Optimalfall davon ausgegangen werden soll, dass keine erneuerbaren Energien ungenutzt bleiben. Das Optimierungsproblem beschränkt sich zunächst auf die Ermittlung der insgesamt anfallenden Treibhausgase.

- Die Zielfunktion minimiert den insgesamten CO2 Ausstoß der Erzeugermenge A:

$$\min \gamma = \min \sum_{\{u \in A | e = uv\}} \left(\sum_t \gamma(u) f_t(u, v) \right) \tag{1.1.1}$$

- Das Flussproblem zum Transport der Energie von den Erzeugern zu den Verbrauchern wird wie folgt über die Kirchhoff'schen Gesetze formuliert:

$$\begin{aligned}
\sum_{e=uv\in E} f_t(u,v) &= p_t^+(u) && \forall t \ \forall u \in A \\
\sum_{e=uv\in E} f_t(u,v) &= p_t^-(v) && \forall t \ \forall v \in B \\
\sum_{e=uv\in E} \eta(u,v) f_t(u,v) - \sum_{e=uv\in E} f_t(v,u) &= 0 && \forall t \ \forall v \in N \setminus \{A,B\}
\end{aligned} \tag{1.1.2}$$

Dabei sind die Flussvariablen $f_t(u,v)$ kapazitätsbegrenzt:

$$0 \leq f_t(u,v) \leq c(u,v) \quad \forall t \ \forall e = uv \in E(G)$$

Das Problem ist linear und kontinuierlich.

Modell 2: Statische Integration von zeitreihengesteuerten Technologien

Im Rahmen einer *statischen Integration* werden konkrete Technologien bereits zum Zeitpunkt der Modellierung platziert und parameterisiert. Die Ansteuerung von Speichern und Konvertern (also deren Verhalten) wird manuell über Zeitreihen (Profile) vorgegeben. Ziel ist die Minimierung der insgesamt anfallenden Treibhausgase aller Erzeuger und aller Technologien, sowie der Vergleich zu den Ergebnissen aus Modell 1.

$$\min \gamma = \left\{ \sum_{\{u\in\{A\cup S\}|e=uv\}} \left(\sum_t \gamma(u) f_t(u,v) \right) + \sum_{e=uv\in C} \gamma(e) f_t(u,v) \right\}$$

Die Gesamtkosten aller integrierten Technologien werden als Nebenbedingung formuliert. Eine obere Schranke k_u definiert die maximalen Ausgaben. Man beachte, dass die linke Seite konstant ist, sodass ein zu gering gewähltes k_u lediglich zur Unlösbarkeit des Optimierungspoblems führen kann:

$$\sum_{v\in S} \left(\hat{k}(v) + \sum_t f_t(u,v) \tilde{k}(v) \right) + \sum_{e=uv\in C} \left(\hat{k}(e) + \sum_t f_t(u,v) \ \tilde{k}(e) \right) \ \leq \ k_u \tag{1.1.3}$$

Die Einbeziehung von in- und exportierten Energiekosten wird äquivalent als Summe formuliert. Anstelle der Summe über die Speichertechnologiemenge S werden entsprechend die Erzeuger- und Verbrauchermengen A und B indexiert, sowie die Fixkosten vernachlässigt.

Die Ansteuerung der Technologien (Einspeicherung, Ausspeicherung, Konvertierung) erfolgt über Zeitreihen. Diese geben für jeden Zeitschritt $t \in [T]$ jeweils den Fluss $f_t(u,v)$ einer jeden Kante aus der Menge $\{e = uv | e \in C \vee v \in S\}$ vor:

$$f_t(u,v) := p_t^+(u,v) := \text{const}$$

Konvertertechnologien, wie z.B. Kraft-Wärme-Kopplungen (KWK) verfügen über einen Eingang und über n Ausgänge. Die Modellierung wird wie in Abb. 1.1.1 gezeigt vorgenommen:

$$f_t(n_1,n_2) = f_t(n1,n_2) = f(n_0,n_1) \cdot \eta(n_0,n_1)$$

Man beachte, dass die Flüsse $f_t(n_1,n_2)$ und $f_t(n_1,n_3)$ gleich sind. Die Effizienzfaktoren $\eta(n_1,n_2)$ und $\eta(n_1,n_3)$ werden *nachgeschaltet* in den Knotengleichungen für n_2 und n_3 berücksichtigt.

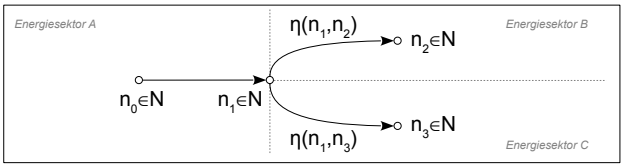

Abb. 1.1.1: Modellierung einer Konvertertechnologie

Modell 3: Statische Integration von optimal gesteuerten Technologien

Die Nutzung von *à priori* definierten Speicher- und Konverterprofilen kann *nicht* für alle Fälle ein globales Optimum hinsichtlich minimaler Treibhausgase liefern. Die Zeitreihen aus Modell 2 werden nun nicht weiter betrachtet. Stattdessen ist der Fluss $f_t(u, v)$ einer jeden Kante aus der Menge $\{e = uv | e \in C \vee v \in S\}$ nun eine Entscheidungsvariable.

Seien $\sigma_t(v)$ nun kontinuierliche Entscheidungsvariablen für den aktuellen Speicherfüllstand (State-of-Charge) des Speichers $v \in S$:

$$\begin{array}{lll} \sigma_0(v) = 0 & & \forall v \in S \\ \sigma_t(v) = \sigma_{t-1}(v) + f_t(u, v) - f_t(v, u) & \forall t \setminus \{0\} & \forall v \in S \\ 0 \leq \sigma_t(v) \leq c(v) & & \forall v \in S \end{array} \tag{1.1.4}$$

Zur drastischen Reduktion der Anzahl an Entscheidungsvariablen kann es (insbesondere bei sehr großen Netzinstanzen) sinnvoll sein, Technologien weiterhin zeitreihengesteuert zu betreiben.

Modell 4: Kontinuierliche Optimierung der Technologieparameter

In den weiteren Betrachtungen sollen die statisch integrierten Technologien optimal dimensioniert werden. Dabei sind $c(v), v \in S$ und $c(e), e \in C$ nun keine Konstanten mehr, sondern Entscheidungsvariablen für die jeweils eine untere und eine obere Schranke definiert wird.

Modell 5: Automatische Intanziierung von Technologien anhand vordefinierter Standortkandidaten

Die Definition von potenziellen Standorten kann *manuell* (Modelle 2 bis 4) oder *algorithmisch* (Modelle 5ff) erfolgen.

Eine algorithmische Ermittlung von Standorten erfordert Ansätze aus der kombinatorischen Optimierung. Folgende Heuristiken schränken die Auswahl potenzieller Lokationen ein:

- Konvertertechnologien können nur dort (zumindest ohne Netzausbau) implementiert werden, wo verschiedene Energiesektoren geometrisch nah beieinander liegen, d.h. wo der euklidische Abstand kleiner als ein gegebenes ϵ ist. Es gelte $u \in V(G_i)$ und $v \in V(G_j)$ für die Sektoren i und j, wobei $i \neq j$:

$$||p(u) - p(v)||_2 < \epsilon$$

- Die Kapazität $c(v)$ einer Speichertechnologie $v \in S$ ist begrenzt durch die inzidierenden Leitungen (Kanten $e \in E(G)$).

Sei I die Menge Standortkandidaten. Die binäre Entscheidungsvariable $I(e = uv) \in \mathbb{Z}_2$ bestimmt, ob eine Technologie tatsächlich instanziiert wird (true = 1), oder nicht (false = 0).

Alle Nebenbedingungen die mit Technologien inzidierende Flussvariablen beinhalten, müssen nun angepasst werden. Dabei wird $f_t(u, v)$ nun durch $I(u, v) \cdot f_t(u, v)$ substituiert.

Modell 6: Automatische Typbestimmung für Technologien

Durch die Komplexität in der realen Welt gestaltet sich das manuelle Festlegen von *Technologietypen* für alle Speicher und Konverter als schwierig, wenn ein *gutes* Mengengerüst gefunden werden soll. Entsprechend wird in diesem Modell die automatische Selektion von Technologietypen aus einer Datenbank in die Problemdefinition einbezogen.

(1.) Speichertechnologien:

Sei $\mathcal{S} = (s_{i,j}) \in \mathbb{R}^{m \times n}$ eine Matrix zur Beschreibung der Datenbank für Speichertechnologien. Jeder Technologietyp wird durch eine Spalte repräsentiert. Spalte j wird definiert durch $s_j = (\eta, c_l, c_u, \gamma, \alpha, \hat{k}_0, \hat{k}_1, \tilde{k}_0, \tilde{k}_1)$. Hierdurch werden für einen Speicher $v \in S$ die folgenden Parameter definiert. Sind einzelne Attribute durch den Wert 0 belegt, so vereinfachen sich die Nebenbedingungen entsprechend.

- $\eta(v) \in [0, 1] \subseteq \mathbb{R}$: Effizienzfaktor.
- $c_l \leq c(v) \leq c_u$: Die Grenzen legen fest, in welchem Spielraum sich die tatsächliche Kapazität bewegen darf.
- $\hat{k}(v) = \hat{k}_1 \cdot c(v) + \hat{k}_0$: Investitionskosten, in Abhängigkeit von der Speicherkapazität.
- $\tilde{k}(v) = \tilde{k}_1 \cdot f_t(u, v) + \tilde{k}_0$: Laufende Kosten, in Abhängigkeit vom Energiefluss.

Die Auswahl einer Technologie erfolgt, pro Standort $v \in S$, durch einen binären Entscheidungsvektor $\lambda \in \mathbb{Z}_2^n$, wobei $\sum_i \lambda_i = 1$. Die Parameterisierung wird per Multiplikation $\mathcal{S} \cdot \lambda$ vorgenommen.

(2.) Konvertertechnologien:

Die Datenbank für Konvertertechnologien wird äquivalent zu der Speicherdefinition als Matrix $\mathcal{C} = (c_{i,j}) \in \mathbb{R}^{m \times n}$ dargestellt.

Modell 7: Erhöhung der physikalischen Genauigkeit

Die bisherigen Modelle implementieren sowohl die Technologien, als auch das Netzverhalten physikalisch eher rudimentär. Da hier auf eine Konkatenation von a) Simulation und b) Optimierung gesetzt wird, fällt das Modell 7 an dieser Stelle weg. Nachteilig bei dieser Methode ist ein erhöhter Rechenaufwand durch *Trial-and-Error*.

1.1.3 Lösung des Optimierungsproblems

Problemklasse

Im Allgmeinen werden für Optimierungsprobleme die folgenden Problemklassen kategorisiert (in der Lineratur findet man weitere Zwischenklassen):

1. Lineare Programmierung, Linear Programming (LP)
2. Quadratische Programmierung, Quadratic Programming (QP)
3. Nichtlineare Programmierung, NonLinear Programming (NLP)
4. Ganzzahlige Programmierung, Integer Programming (IP)
5. Gemischt Ganzzahlige Programmierung, Mixed-Integer Linear Programming (MILP)
6. Gemischt Ganzzahlige Nichtlineare Programmierung, Mixed-Integer NonLineare Programming (MINLP)

Als Sonderfall von 4.-6. können die diskreten Anteile rein binär sein, sodass dann von einer 0-1 Programmierung gesprochen werden kann.

Die folgende Tabelle zeigt die jeweilige Problemklasse aus den oben beschrieben Modellen.

Modelle	Problemklasse
1-4	Lineare Programmierung (LP)
4	Quadratische Programmierung (QP)
5-6	0-1 Gemischt-Ganzzahlige Quadratische Programmierung (MIQCP)
7	0-1 Gemischt-Ganzzahlige Nichtlineare Programmierung (MINLP)

Um alle möglichen und quantitativ komplexen Eingabeszenarien abzudecken, muss ein Löser für die Problemklasse MINLP herangezogen werden:

$$\min_{x,y} f(x,y) \;\; s.t.\; g(x,y) \leq 0$$

Mit $f : \mathbb{R}^n \to \mathbb{R}$, $g : \mathbb{R}^n \to \mathbb{R}^m$, $x \subseteq \mathbb{R}^{n_1}, y \subseteq \mathbb{Z}^{n_2} (n := n_1 + n_2)$.

I.d.R. sind alle Probleme mit ganzzahligem Anteil nur sehr schwer lösbar, gehören also der Komplexitätsklasse NP an.

Löser

Für die Lösung von Problem der Klasse *Mixed-Integer NonLinear Programs* (MINLP) ist auf dem Markt eine Vielzahl an Solvern verfügbar. Im Rahmen von ES-Flex-Infra sollen *freie* und *quelloffene* Solver Anwendung finden. Insbesondere die in den nächsten Abschnitten genutzten Löser kommen dabei in Betracht. Im Kapitel Software wird die konkrete Integration der Solver in das System beschrieben.

Ipopt

IPOPT wird bereits von SCAI für Optimierungsprobleme aus der Klasse NLP (u.a.) im Rahmen des Tools MYNTS eingesetzt (vgl. mit dem Kapitel Software).

BONMIN (Basic Open-source Nonlinear Mixed Integer Programming)

Da Löser i.d.R. keine globalen Extrema finden, weist das kontextsensitive Verhalten bezogen auf die konkreten Eingabedaten gewisse Charakteristiken (Qualität, Genauigkeit, ...) auf. Die Entwicklung der von SCAI entwickelten Tools hat (für andere Projekte) bereits vor Projektstart begonnen. Entsprechend ist eine Rücksichtnahme auf die bestehenden Softwarestrukturen essentiell. Eine Migration von *BONMIN* kann aufgrund der ähnlichen Programmierschnittstelle zur *IPOPT* relativ einfach vorgenommen werden. Weiterhin kann bei gleicher Solverkonfiguration für NLP-Probleme eine numerische Äquivalenz zwischen den Tools nachgewiesen werden. Entsprechend bleiben nach Migration von BONMIN auch bestehende Testfälle usw numerisch gültig.

Eine ausfühliche Diskussion für die Wahl des Lösers ist im Dokument „Mathematische Optimierung im Projekt ES-Flex-Infra“ (siehe Projektdatenarchiv) ausgegliedert. Darin werden auch die Pakete *Couenne*, *LaGO* und *SCIP* diskutiert.

Zusammenfassend führen die folgenden Gesichtspunkte zur Wahl von BONMIN:

- Kompatibilität zu *IPOPT*.
- Open Source, und damit das Potenzial selbst Änderungen am Quelltext vornehmen zu können.
- Eclipse Public License: private und kommerzielle Nutzung erlaubt (Achtung: der Sourcecode von BONMIN muss bei einer Distribution wieder mit ausgeliefert werden!).
- Große Community (COIN-OR).

- Schnittstellenvielfalt: COIN-OR, AMPL, GAMS. Weiterhin ist eine native Nutzung von BONMIN als Bibliothek über die Programmiersprache C++ möglich.
- Auswahlmöglichkeit diverser interner Algorithmen, sodass abhängig vom Eingabeproblem das Konvergenzverhalten verbessert werden kann.
- Die unterliegenden Subsolver/Bibliotheken sind konfiguierbar und teilweise substituierbar, sodass z.B. der LP-Solver bei Bedarf durch ein kommerzielles Tool ersetzt werden kann.

Nachteilig gegenüber anderen Lösern sind u.a.:

- Optima sind lokal (ebenso wie bei IPOPT).
- BONMIN ist als freie Solver für die meisten Eingabedaten weniger performant als kommerzielle Solver.

Gurobi

Da die Berechnungszeit von BONMIN bei erweiterten Probleminstanzen sehr groß ist, wurde auch der Löser *Gurobi* in die Implementierung testweise eingebunden.

1.1.4 Testfälle

Die folgenden einfachen Testfälle evaluieren das System und bedienen sich dabei in der Modellierung an der Domänenspezifischen Sprache CEMPL. Diese wird im Kapitel Software ausführlich beschrieben. Auf die Angabe der Zwischenausgaben in Form einer Intermediate Language (IM) wird hier verzichtet. Man beachte, dass die definierten Werte hypothetisch sind, um den konkreten *Softwaretests(!)* dienlich zu sein.

Modell 1 (Flußproblem ohne Technologien)

Abb. 1.1.2 zeigt ein einfaches Netz, bestehend aus je einem Knoten für einen Erzeuger und einem Verbraucher. Die Last- und Einspeiseprofile umfassen je drei Zeitschritte. Die Leitung ist verlustlos.

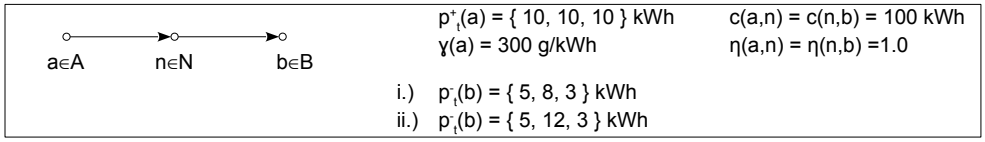

Abb. 1.1.2: Test 1: 1 Erzeuger / 1 Verbraucher

Code Block 1.1.1: test1.mod

```
energytype power:
        unit = kWh
producer a:
        profile = [10, 10, 10]
        co2 = 300 [g_kWh]
consumer b:
        profile = [5, 8, 3]    % Test i.)
        profile = [5, 12, 3]   % Test ii.)
network net:
        producer a: x= 0, y=0
        node     n: x= 5, y=0
        consumer b: x=10, y=0
        a -> n: capacity=100
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        n -> b: capacity=100
scenario s1:
        use network net
```

Tabelle 1.1.1: Lösung zu Abb. 1.1.2

i.)	$f_t(a,n) = f_t(n,b) = p_t^- = \{5, 8, 3\}$ kWh $\sum_t \gamma = 4800$ g/kWh
ii.)	keine zulässige Lösung

Abb. 1.1.3 erweitert den letzten Test um einen zusätzlichen Erzeuger mit höheren Treibhausgasen.

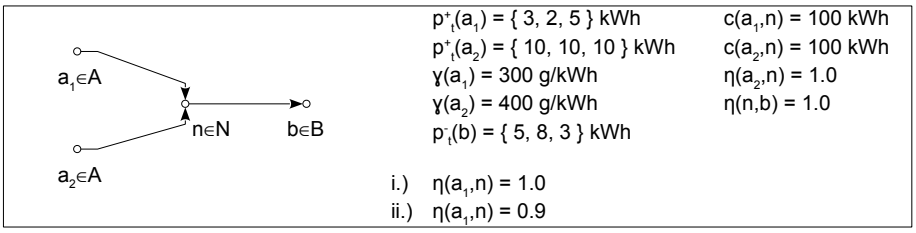

Abb. 1.1.3: Test 2

Code Block 1.1.2: test2.mod

```
energytype power:
        unit = kWh
producer a1:
        profile = [3, 2, 5]
        co2 = 300 g_kWh
producer a2:
        profile = [10, 10, 10]
        co2 = 400 g_kWh
consumer b:
        profile = [5, 8, 3]
network net:
        producer a1: x= 0, y= 0
        producer a2: x= 0, y=10
        node     n:  x= 5, y= 5
        consumer b:  x=10, y= 5
        line: a1 -> n: capacity=100, eta=1.0
        line: a2 -> n: capacity=100
        line: n  -> b: capacity=100
scenario s1:
        use network net
options:
        solver = gurobi
```

Modell 2 (Statische Integration von zeitreihengesteuerten Technologien)

In Abb. 1.1.4 wird ein profilgesteuerter Speicher integriert.

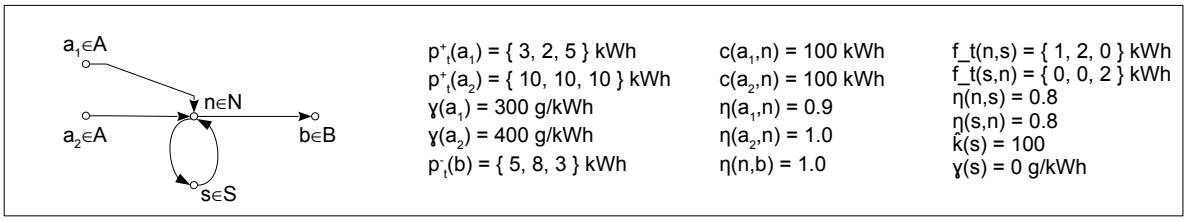

Abb. 1.1.4: Test 3a

Code Block 1.1.3: test3a.mod

```
energytype power:
        unit = kWh
producer a1:
        profile = [3, 2, 5]
        co2 = 300 g_kWh
producer a2:
        profile = [10, 10, 10]
        co2 = 400 g_kWh
consumer b:
        profile = [5, 8, 3]
storage s:
        energytype = power
        capacity = 3 kWh
        investment_costs:
                base = 100 EUR
                per_kWh = 20 EUR
        input:
                eta = 0.8
                max_flow = 2.01 kWh
        output:
                eta = 0.8
                max_flow = 2.02 kWh
        profile = [1, 2, -2]
network net:
        producer a1: x= 0, y= 0
        producer a2: x= 0, y= 5
        node     n:  x= 5, y= 5
        consumer b:  x=10, y= 5
        storage  s:  x= 5, y=10
        line: a1 -> n: capacity=100, eta=0.9
        line: a2 -> n: capacity=100
        line: n  -> b: capacity=100
        line: n <-> s
scenario s1:
        use network net
options:
        solver = gurobi
```

In Abbildung Abb. 1.1.5 wird ein profilgesteuerter Konverter dargestellt.

Code Block 1.1.4: test3b.mod

```
energytype power:
```

(Fortsetzung auf der nächsten Seite)

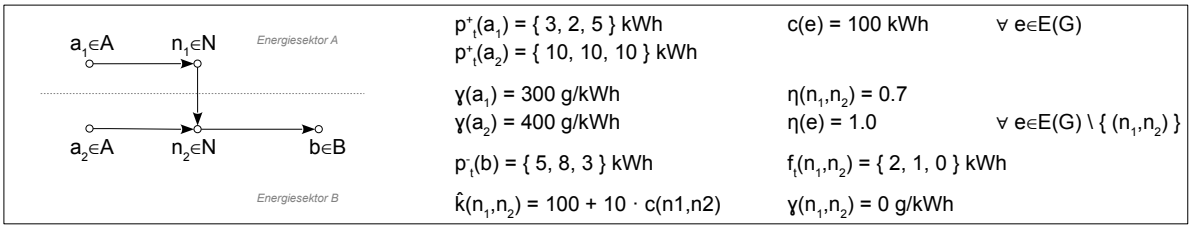

Abb. 1.1.5: Test 3b

(Fortsetzung der vorherigen Seite)

```
        unit = kWh
energytype gas:
        unit = kWh
producer a1:
        profile = [3, 2, 5]
        co2 = 300 g_kWh
producer a2:
        profile = [10, 10, 10]
        co2 = 400 g_kWh
consumer b:
        profile = [5, 8, 3]
converter c:
        investment_costs:
                base = 100 EUR
                per_kWh = 10 EUR
        max_flow = 3 kWh
        profile = [2, 1, 0]
        input:
                energytype = power
        output:
                energytype = gas
                eta = 0.7
network net:
        producer a1: x= 0, y=0
        producer a2: x= 0, y=5
        node     n1: x= 5, y=0
        node     n2: x= 5, y=5
        consumer b:  x=10, y=5
        line:        a1 -> n1: capacity=100, eta=0.9
        line:        a2 -> n2: capacity=100
        line:        n2 -> b:  capacity=100
        converter c: n1 -> n2
scenario s1:
        use network net
options:
        solver = gurobi
```

Modell 3 (Statische Integration von optimal gesteuerten Technologien)

In Abb. 1.1.6 wird ein Speicher integriert, dessen Laufzeitverhalten nicht als Zeitreihe definiert ist. Stattdessen ist die Bestimmung der zeitabhängigen optimalen Ein- und Ausspeisung teil des Optimierungsproblems.

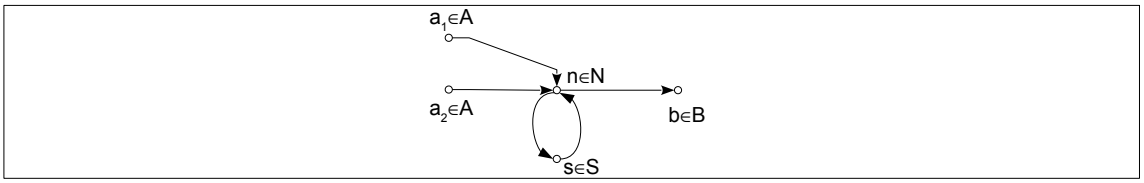

Abb. 1.1.6: Test 4a

Code Block 1.1.5: test4a.mod

```
energytype power:
        unit = kWh
producer a1:
        profile = [8, 5, 0]
        co2 = 50 g_kWh
producer a2:
        profile = [10, 10, 10]
        co2 = 400 g_kWh
consumer b:
        profile = [5, 8, 3]
storage s:
        energytype = power
        capacity = 3 kWh
        investment_costs:
                base = 100 EUR
                per_kWh = 20 EUR
        input:
                eta = 0.8
                max_flow = 3 kWh
        output:
                eta = 0.8
                max_flow = 3 kWh
network net:
        producer a1: x= 0, y= 0
        producer a2: x= 0, y= 5
        node     n:  x= 5, y= 5
        consumer b:  x=10, y= 5
        storage  s:  x= 5, y=10
        line: a1 -> n: capacity=100, eta=0.9
        line: a2 -> n: capacity=100
        line: n  -> b: capacity=100
        line: n <-> s
scenario s1:
        use network net
options:
        solver = gurobi
```

In Abb. 1.1.7 wird ein Konverter integriert, dessen Laufzeitverhalten nicht als Zeitreihe definiert ist. Stattdessen ist die Bestimmung der zeitabhängigen Durchflusses teil des Optimierungsproblems.

Code Block 1.1.6: test4b.mod

```
energytype power:
        unit = kWh
energytype gas:
```

(Fortsetzung auf der nächsten Seite)

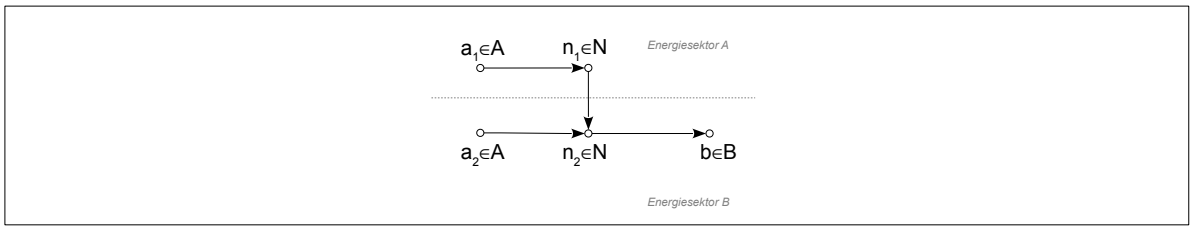

Abb. 1.1.7: Test 4b

(Fortsetzung der vorherigen Seite)

```
        unit = kWh
producer a1:
        profile = [3, 2, 5]
        co2 = 50 g_kWh
producer a2:
        profile = [10, 10, 10]
        co2 = 400 g_kWh
consumer b:
        profile = [5, 8, 3]
converter c:
        investment_costs:
                base = 100 EUR
                per_kWh = 10 EUR
        max_flow = 3 kWh
        input:
                energytype = power
        output:
                energytype = gas
                eta = 0.7
network net:
        producer a1: x= 0, y=0
        producer a2: x= 0, y=5
        node     n1: x= 5, y=0
        node     n2: x= 5, y=5
        consumer b:  x=10, y=5
        line:        a1 -> n1: capacity=100, eta=0.9
        line:        a2 -> n2: capacity=100
        line:        n2 -> b:  capacity=100
        converter c: n1 -> n2
scenario s1:
        use network net
options:
        solver = gurobi
```

Modell 4 (Kontinuierliche Optimierung der Technologieparameter)

Das Speicherszenario wird nun dahingehend erweitert, dass die Dimensionierung des Speichers als Teil des Optimierungsproblems mitgelöst wird.

Code Block 1.1.7: test5a.mod

```
energytype power:
        unit = kWh
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
producer a1:
        profile = [8, 5, 0]
        co2 = 50 g_kWh
producer a2:
        profile = [10, 10, 10]
        co2 = 400 g_kWh
consumer b:
        profile = [5, 8, 3]
storage s:
        energytype = power
        capacity = 0.5 .. 3 kWh
        investment_costs:
                base = 100 EUR
                per_kWh = 20 EUR
        input:
                eta = 0.81
                max_flow = 0.3 kWh
        output:
                eta = 0.82
                max_flow = 0.3 kWh
network net:
        producer a1: x= 0, y= 0
        producer a2: x= 0, y= 5
        node     n:  x= 5, y= 5
        consumer b:  x=10, y= 5
        storage  s:  x= 5, y=10
        line: a1 -> n: capacity=100, eta=0.9
        line: a2 -> n: capacity=100
        line: n  -> b: capacity=100
        line: n <-> s
scenario s1:
        use network net
        limit costs to 110 EUR
options:
        solver = gurobi
```

Das folgende Beispiel bestimmt die Dimensionierung eines Energiekonverters per Optimierung.

Code Block 1.1.8: test5b.mod

```
energytype power:
        unit = kWh
energytype gas:
        unit = kWh
producer a1:
        profile = [3, 2, 5]
        co2 = 50 g_kWh
producer a2:
        profile = [10, 10, 10]
        co2 = 400 g_kWh
consumer b:
        profile = [5, 8, 3]
converter c:
        investment_costs:
                base = 100 EUR
                per_kWh = 10 EUR
        max_flow = 0.5 .. 3 kWh
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        input:
                energytype = power
        output:
                energytype = gas
                eta = 0.7
network net:
        producer a1: x= 0, y=0
        producer a2: x= 0, y=5
        node     n1: x= 5, y=0
        node     n2: x= 5, y=5
        consumer b:  x=10, y=5
        line:        a1 -> n1: capacity=100, eta=0.9
        line:        a2 -> n2: capacity=100
        line:        n2 -> b:  capacity=100
        converter c: n1 -> n2
scenario s1:
        use network net
        limit costs to 125 EUR
options:
        solver = gurobi
```

Modell 5 (Automatische Instanziierung von Technologien anhand vordefinierter Standortkandidaten)

In Abb. 1.1.8 stehen zwei potenzielle Speicherlokation zur Auswahl. Es gilt zu ermitteln, ob keiner, einer oder zwei Speicher implementiert werden sollten.

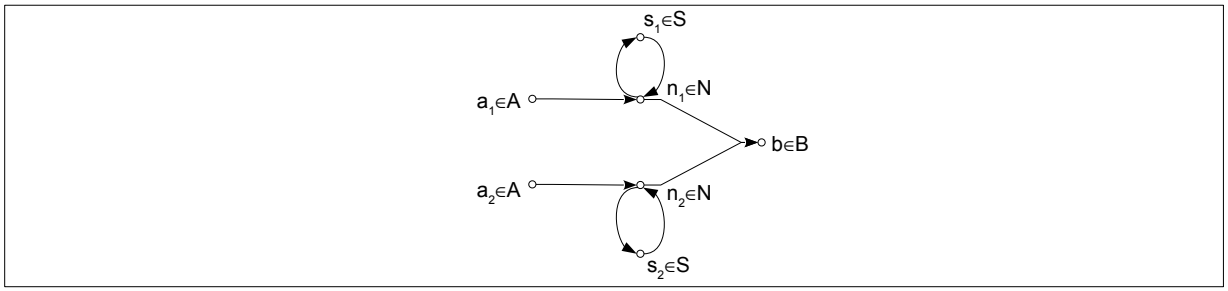

Abb. 1.1.8: Test 6a

Code Block 1.1.9: test6a.mod

```
energytype power:
        unit = kWh
producer a1:
        profile = [50, 50, 50]
        co2 = 300 g_kWh
producer a2:
        profile = [15, 0, 3]
        co2 = 50 g_kWh
consumer b:
        profile = [5, 8, 10]
storage s:
        energytype = power
        capacity = 3 kWh
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        investment_costs:
                base = 100 EUR
                per_kWh = 20 EUR
        input:
                eta = 0.8
                max_flow = 1.25 kWh
        output:
                eta = 0.8
                max_flow = 1.25 kWh
network net:
        producer a1: x= 0, y= 0
        producer a2: x= 0, y= 5
        node     n1: x= 5, y= 0
        node     n2: x= 5, y= 5
        consumer b:  x=10, y= 2.5
        optional storage s as s1:  x= 5, y=-5
        optional storage s as s2:  x= 5, y=10
        line: a1 -> n1: capacity=100, eta=0.9
        line: a2 -> n2: capacity=100
        line: n1  -> b: capacity=100
        line: n2  -> b: capacity=100
        line: n1 <-> s1
        line: n2 <-> s2
scenario s1:
        use network net
        limit costs to 200 EUR
options:
        solver = gurobi
```

Im folgenden Code gilt es zu ermitteln, ob ein Konverter instanziiert werden soll.

Code Block 1.1.10: test6b.mod

```
energytype power:
        unit = kWh
energytype gas:
        unit = kWh
producer a1:
        profile = [3, 2, 5]
        co2 = 50 g_kWh
producer a2:
        profile = [10, 10, 10]
        co2 = 400 g_kWh
consumer b:
        profile = [5, 8, 3]
converter c:
        investment_costs:
                base = 100 EUR
                per_kWh = 10 EUR
        max_flow = 3 kWh
        input:
                energytype = power
        output:
                energytype = gas
                eta = 0.7
network net:
        producer a1: x= 0, y=0
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        producer a2: x= 0, y=5
        node     n1: x= 5, y=0
        node     n2: x= 5, y=5
        consumer b:  x=10, y=5
        line:        a1 -> n1: capacity=100, eta=0.9
        line:        a2 -> n2: capacity=100
        line:        n2 -> b:  capacity=100
        optional converter c as c: n1 -> n2
scenario s1:
        use network net
        limit costs to 125 EUR
options:
        solver = gurobi
```

Modell 6 (Automatische Typbestimmung für Technologien)

In Abb. 1.1.9 ist der Speichertyp zunächst undefiniert. Die Lösung des Optimierungsproblems umfasst die Auswahl des geeignetsten Speichertyps aus der definierten Gerätedatenbank.

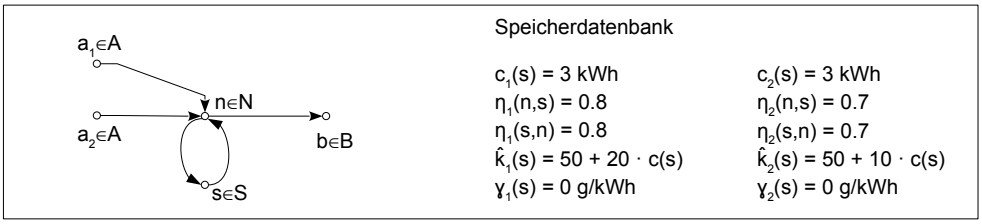

Abb. 1.1.9: Test 7a

Code Block 1.1.11: test7a.mod

```
energytype power:
        unit = kWh
producer a1:
        profile = [8, 5, 0]
        co2 = 50 g_kWh
producer a2:
        profile = [10, 10, 10]
        co2 = 400 g_kWh
consumer b:
        profile = [5, 8, 3]
storage s1:
        energytype = power
        capacity = 2 kWh
        investment_costs:
                base = 100 EUR
                per_kWh = 25 EUR
        input:
                eta = 0.81
                max_flow = 0.96 kWh
        output:
                eta = 0.83
                max_flow = 0.98 kWh
storage s2:
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        energytype = power
        capacity = 3 kWh
        investment_costs:
                base = 90 EUR
                per_kWh = 20 EUR
        input:
                eta = 0.82
                max_flow = 0.97 kWh
        output:
                eta = 0.84
                max_flow = 0.99 kWh
network net:
        producer a1: x= 0, y= 0
        producer a2: x= 0, y= 5
        node     n:  x= 5, y= 5
        consumer b:  x=10, y= 5
        storage s1 or s2 as s:  x= 5, y=10
        line: a1 -> n: capacity=100, eta=0.9
        line: a2 -> n: capacity=100
        line: n  -> b: capacity=100
        line: n <-> s
scenario s1:
        use network net
        limit costs to 200 EUR
options:
        solver = gurobi
```

Die systemgenerierte Lösung ist in Abb. 1.1.10 zu sehen.

Nun soll der Typ eines Konverters per Optimierung bestimmt werden:

Code Block 1.1.12: test7b.mod

```
energytype power:
        unit = kWh
energytype gas:
        unit = kWh
producer a1:
        profile = [3, 2, 5]
        co2 = 50 g_kWh
producer a2:
        profile = [10, 10, 10]
        co2 = 400 g_kWh
consumer b:
        profile = [5, 8, 3]
converter c1:
        investment_costs:
                base = 100 EUR
                per_kWh = 10 EUR
        max_flow = 3 kWh
        input:
                energytype = power
        output:
                energytype = gas
                eta = 0.7
```

(Fortsetzung auf der nächsten Seite)

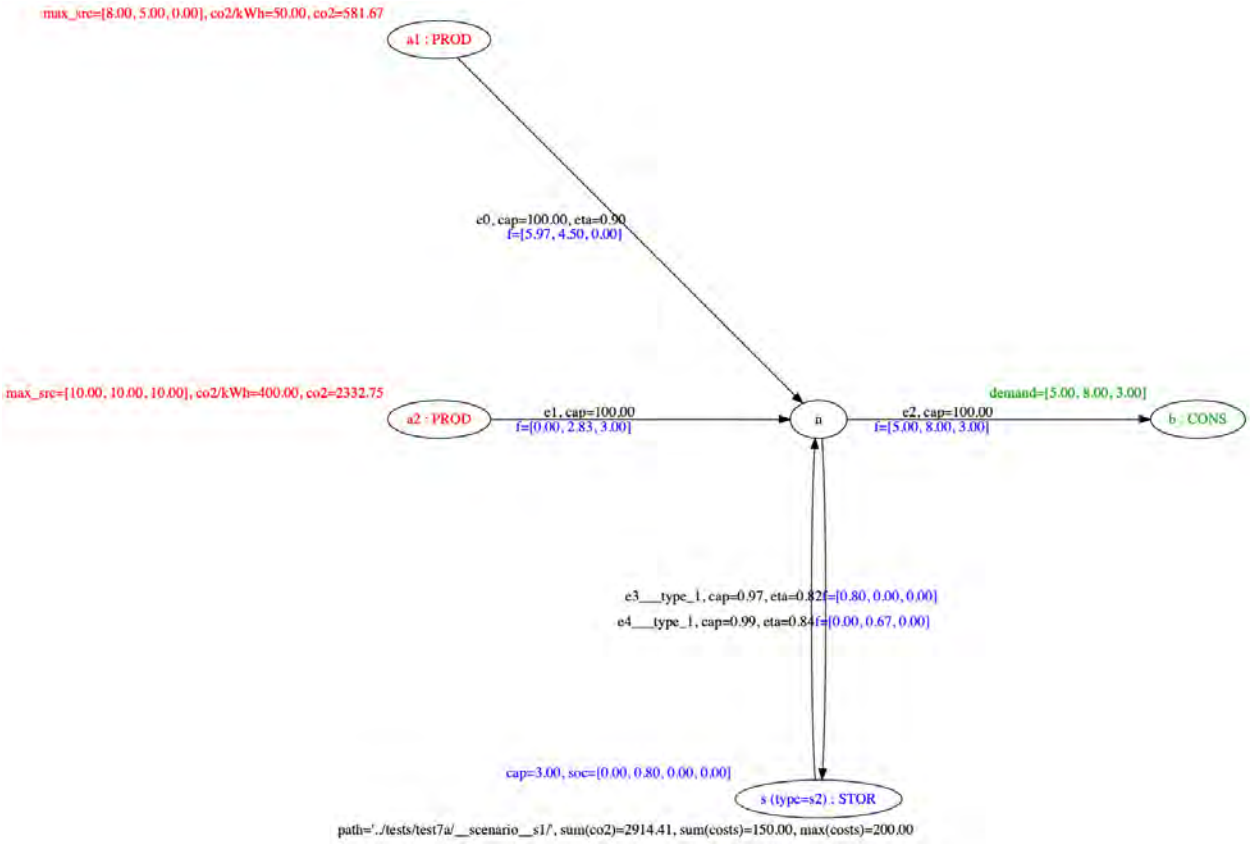

Abb. 1.1.10: Lösung Test 7a

(Fortsetzung der vorherigen Seite)

```
converter c2:
        investment_costs:
                base = 50 EUR
                per_kWh = 5 EUR
        max_flow = 3 kWh
        input:
                energytype = power
        output:
                energytype = gas
                eta = 0.9
network net:
        producer a1: x= 0, y=0
        producer a2: x= 0, y=5
        node     n1: x= 5, y=0
        node     n2: x= 5, y=5
        consumer b:  x=10, y=5
        line:        a1 -> n1: capacity=100, eta=0.9
        line:        a2 -> n2: capacity=100
        line:        n2 -> b:  capacity=100
        converter c1 or c2 as c: n1 -> n2
scenario s1:
        use network net
        limit costs to 125 EUR
options:
        solver = gurobi
```

Die systemgenerierte Lösung wird in Abb. 1.1.10 gezeigt.

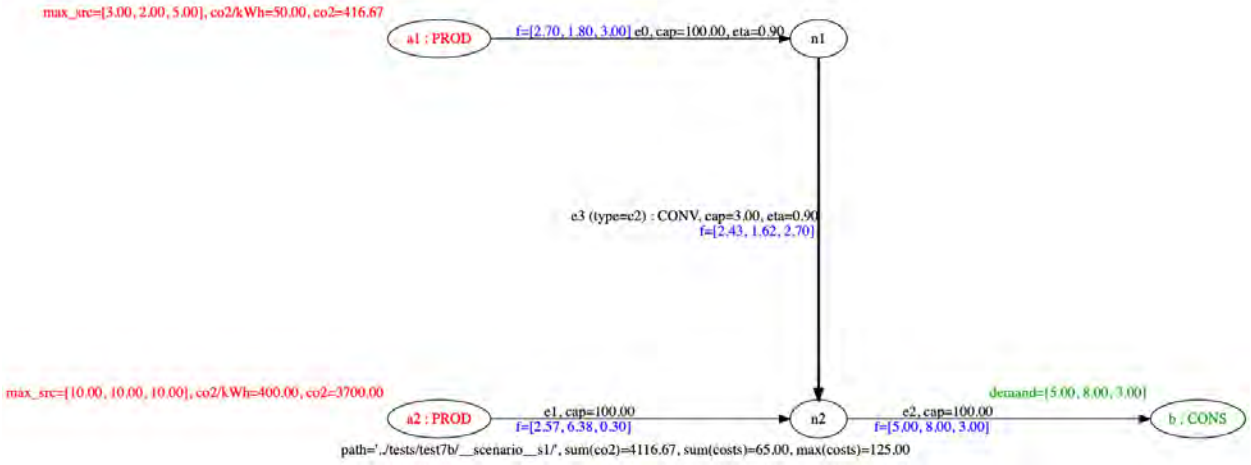

Abb. 1.1.11: Lösung Test 7b

KAPITEL 2

Software

Eines der Kernziele des Projekts ist die Schaffung einer Softwarebasis, die dazu in der Lage ist die in den vorherigen Kapiteln definierten Ziele umzusetzen. Dieses Kapitel beschreibt die Softwarearchitektur, die einzelnen Softwarekomponenten, sowie die unterliegenden Datenbanken.

2.1 Softwarearchitektur

Die im Projekt entwickelte und eingesetzte Softwarearchitektur wird in Abb. 2.1.1 gezeigt. Die mittig dargestellten Blöcke sind als Softwarekomponenten (Programme) zu verstehen. Die abgerundeten Blöcke an dem oberen und unteren Rand repräsentieren Datenbanken.

2.1.1 Kurzbeschreibung der Komponenten

Das System besteht aus einer Menge von Softwarekomponenten, die in diesem Abschnitt kurz umrissen werden. Die Systemintegration wird im nächsten Abschnitt beschrieben. Durch das Konzept der *losen Kopplung* werden die Einzelbausteine so entworfen, dass diese auch möglichst isoliert voneinander arbeiten können (Ausnahme: Komponente „Wrapper“). Ziel ist es, die Wiederverwendbarkeit der einzelnen Pakete der Projektpartner zu gewährleisten.

Detailliertere Beschreibungen für jede Komponente findet man in den nachfolgenden Unterkapiteln.

MYNTS (MultiphYsical NeTwork Simulator)

Durch Fraunhofer SCAI erstelltes Softwarepaket zur Planung und Simulation komplexer Energienetze. Im hiesigen Projekt werden primär die Komponenten für die Berechnung von Gas- und Fernwärmenetzen eingesetzt. Für Simulationsläufe wird per mathematischer Programmierung (in Form von nichtlinearen Programmen; NLP) Probleminstanzen erzeugt, die mittels des freien Solvers *Ipopt* gelöst werden.

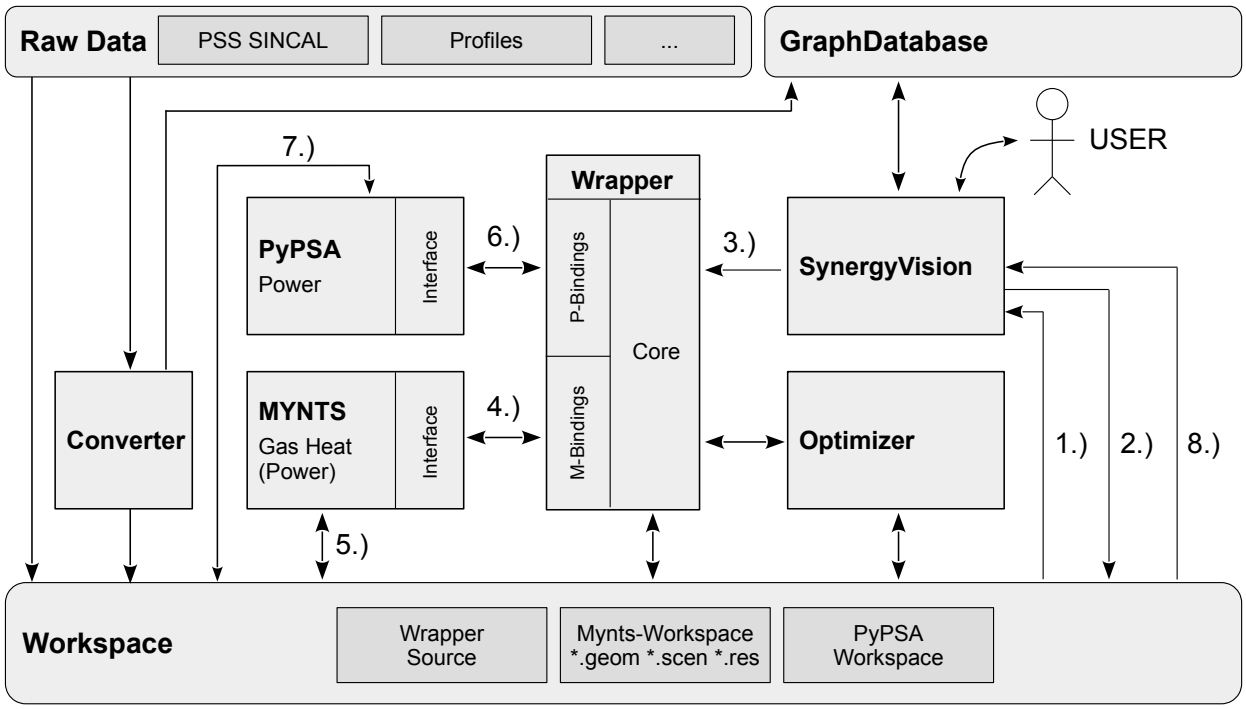

Abb. 2.1.1: Softwarearchitektur

PyPSA (Python for Power System Analysis)

PyPSA ist eine freie und quelloffene Software zur Simulation und Optimierung von Stromnetzen.

Converter

Durch die *werusys GmbH* entwickeltes Daten-Konvertierungstool. Unter anderem werden die im Format *PSS Sincal* vorliegenden Netzdaten hin zu *Mynts* kompatiblen Formaten transformiert.

SynergyVision

SynergyVision ist ein Produktionsmanagementsystem (Manufacturing Execution System, MES) der *werusys GmbH*. Durch Zugriff auf Produktionsdaten werden Informationen zur Effizienz, Auslastung, Mengenbilanzen und Verfügbarkeiten für die Produktionsoptimierung visualisiert. In diesem Projekt werden vor allem die Frontend-Eigenschaften genutzt.

Optimizer

Die Komponente *Optimizer* erzeugt ein Mengengrüst an Energiespeicher- und Energiekonvertertechnologien. Die insgesamt im Netz anfallenden Treibhausgase (insbesondere erzeugerseitig) werden minimiert. Die Berechnung stützt sich auf approximierten Netzdaten. Die realen physischen Eigenschaften werden abstrahiert, um auch bei quantitativ sehr großen Probleminstanzen die Lösbarkeit zu gewährleisten. Per Definition von formalen Schnittstellen wird ein valider Berechnungsworkflow ermöglicht. Eine Verifikation der durch die Optimierung erzeugten Postulate erfolgt durch die *Simulationskomponenten* (siehe *Mynts*, *PyPSA* und *Wrapper*).

Eine erweiterte Softwarearchitektur mit den Subkomponenten des Optimizer wird in der folgenden Abbildung dargestellt:

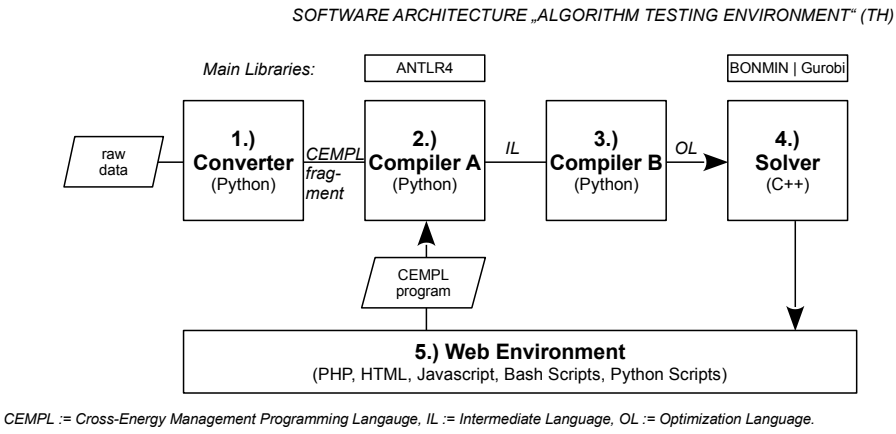

Abb. 2.1.2: Softwarearchitektur Optimierung

Kurzbeschreibung:

1. (Optionale) Konvertierung von Rohdaten in Codefragmente der Sprache CEMPL.
2. Kompilierung von CEMPL-Code in die Zwischensprache IL (Intermediate Language).
3. Erzeugung von Code in der Zwischensprache OL (Optimization Language) für den Optimierer.
4. Mathematische Optimierung; gestützt durch den Löser BONMIN.
5. (Optionale) Webumgebung zur Ansteuerung der Komponenten 1-4.

Wrapper

Die in der Programmiersprache *Python* geschriebene Komponente *Wrapper* dient der globalen Steuerung des Simulations- und Optimierungsablaufs. Sie stellt ein Bindeglied dar, das die einzelnen Komponenten in Interaktion bringt.

2.1.2 Kurzbeschreibung der Datenbanken

Im Rahmen des Projekts wird u.a. die folgende Datenbasis verarbeitet:

- Netzdaten: Graphen mit physikalischen Attributen
- Physikalische Beschreibungen von Speicher- und Konvertertechnologien
- Zeitreihen: Einspeiseprofile und Lastflüsse
- monetäre Kenngrößen: Investitionskosten und laufende Kosten
- Treibhausgase
- usw

Diese Datenbestände werden in den folgenden Datenbanken persistent abgelegt:

RNG Daten

Die von der *Rheinischen NETZGesellschaft* bereitgestellten Daten liegen weitestgehend als unverarbeitete Rohdaten vor. Beispielformate sind Netzdaten in *PSS SINCAL*, Zeitreihen in Form von *Comma-Separated Value (CSV)* Beständen und *Excel-Tabellen*.

Workspace

Der *Workspace* besteht aus einer vorverarbeiteten Datenbasis und bildet die Grundlage für den Simulations- und Optimierungsablauf. Die Komponente *Converter* erstellt diesen Datenbestand aus den Rohdaten.

Graphdatabase

Für die graphische Repräsentation in der Komponente *SynergyVision* liegen die Netzdaten in Form einer Graphdatenbank (Neo4j) vor.

2.1.3 Exemplarische Ablaufbeschreibung

Die Beschreibung des Zusammenspiels der einzelnen Softwarekomponenten aus Abb. 2.1.1 wird nun durch einen (einfachen) Anwendungsfall motiviert:

Der Benutzer fügt dem aktuell betrachteten Energienetz eine zusätzliche KWK-Anlage hinzu. Datengrundlage ist das im Netzkapitel beschriebene *Testnetz*. Zur Ansteuerung der Speicher- und Konvertertechnologie stehen Zeitreihen bereit. Das Energienetz wird hier zunächst manuell und statisch erweitert. Eine Optmierung hinsichtlich der Technologieparameter findet in dem beschrieben Anwendungsfall zunächst nicht statt.

Die einzelnen Schritte beziehen sich auf die Nummerierung der Kanten aus Abb. 2.1.1.

1. Einlesen der Konfigurationsdatei (`config.json`). Darin sind u.a. die Verzeichnispfade des aktuell betrachteten Szenarios enthalten. Entsprechend kann das Netz aus der Graphdatenbank geladen und über den Webservice von *SynergyVision* visualisiert werden.
2. Der Benutzer (USER) fügt nun eine weitere Konvertertechnologie (hier: KWK Anlage) hinzu. In den Workspace werden nun die entsprechenden formalen Beschreibungen für die Simulatoren geschrieben. Für die Sektoren Gas und Fernwärme werden *Mynts*-Datenbetände in Form von `*.geom` und `*.scen` Dateien generiert. Für den Sektor Strom werden *PyPSA*-Datenbestände in Form von `*.csv` Dateien generiert.
3. Die Benutzeroberfläche in *SynergyVision* startet den Simulationsablauf über die Softwarekomponente *Wrapper*. Für jeden simulierten Zeitschritt werden die (unten stehenden) Schritte 4 bis 7 ausgeführt. Die Anzahl an Zeitschritten wird durch die Länge der Zeitreihen (Einspeise-, Last-, Speicher- und Konverterprofile) bestimmt. Die einzelnen Simulatoren (*Mynts* und *PyPSA*) nutzen die Datenbank *Workspace* als Datengrundlage. Sektorübergreifend werden Zwischenergebnisse über die Schnittstellen kommuniziert. Die Komponente *Wrapper* hält stets alle Zwischenergebnisse über eine Menge an Tripeln in der Form $(attribute, value, timestamp)$ im Arbeitsspeicher, sodass auch zeit- und ortsübergreifende Regeln (z.B. zur Speicheransteuerung) implementiert werden können.
4. Der Gas- und Fernwärmesimulator *Mynts* berechnet den nächsten Zeitschritt $t+1$. Die Ergebnisse der aktuellen Iteration werden an die Komponente *Wrapper* zurückgegeben.
5. *Mynts* speichert die Berechnungsergebnisse des aktuellen Zeitschritts in den *Workspace* (in je einen neuen Ordner innerhalb der Verzeichnisstruktur).
6. Der Stromsimulator *PyPSA* berechnet den nächsten Zeitschritt. Die Berechnungsergebnisse werden an die Komponente *Wrapper* zurückgegeben.
7. *Mynts* speichert die Berechnungsergebnisse des aktuellen Zeitschritts in den *Workspace*.

8. Nach Beendigung der Simulationsschritte liest *SynergyVision* die Resultate ein und bereitet diese dem Benutzer (USER) grafisch (u.a. in Form von Lastflüssen) auf.

2.1.4 Software-Verzeichnisstruktur (TH-Opt.)

Die folgenden Verzeichnisse sind im Projekt-Softwarearchiv (GIT) verfügbar. Darunter finden sich Kernkomponenten, sowie teils partiell implementierte Hilfskomponenten, die für Software-Tests usw. aufgebaut wurden.

- **compiler/** „Compiler A“: Übersetzt die domänenspezifische Sprache CEMPL (Cross Energy Management Programming Language) in einen Zwischencode IM (Intermediate Language).
- **optimizer/** „Compiler B“ (der Projektname „optimizer“ ist über die Projektlaufzeit entfremdet) zur Übersetzung von IM-Code in ein mathematisches Optimierungsproblem (Optimization Langauge, OL).
- **solver/** In C++ geschriebene Einbettungsumgebung für den Solver BONMIN. Enthält Code für die symbolische Behandlung von mathematischen Audrücken (Termen), inklusive mehrfacher-automatischer partieller Abteilungen usw.
- **web/** Webservice zur Steuerung der Algorithmentestplattform.
- **userdata/** Nutzerdaten des Webservice.
- **tests/** Testfälle in Form von CEMPL-Modellen. Auch die Zwischenergebnisse, Plots, wertmäßige Ausgaben, ... sind enthalten.
- **../wrapper/** Wrapperkomponente zur globalen Ablaufsteuerung im Gesamtprojekt.
- **converter/** Importiert einen MYNTS-Workspace und exportiert (rudimentär) CEMPL-Modelle.
- **data/** Datenextrakte aus dem Projekt.
- **forecast/** (Rudimentäre) Tests für das Aufstellen von Forecasts druch AR(I)MA Modelle (z.B: saisonale Auswertungen)
- **synthesizer/** Rudimentäre Implementierung eines Zeitreihengenerators.

2.2 Wrapper

Die Komponente *Wrapper* ist eine Integrationskomponente und dient als Steuerungs- und Bindeglied zwischen den Softwarekomponenten des Gesamtprojekts. Als lose gekoppelte Softwarearchitektur werden die weitgehend autark arbeitenden Softwarebauteine per definierten Kommunikationsschnittstellen zusammengefügt. Die Komponente steuert vor allem den simulativen Berechnungs- und Optimierungsworkflow des Gesamtsystems. Dazu gehört zum Beispiel die Überwachung des Konvergenzverhaltens.

Der Entwurf bedient sich des Konzepts der „losen Kopplung“, sodass die im Projekt entstandenen Softwarekomponenten durch wohldefinierte Interfaces in Interaktion gebracht und bei Bedarf flexibel durch mächtigere Bausteine substituiert werden können.

Designziele:

- Hierarchischer Entwurf: Zum Beispiel ist eine reine Simulationsfähigkeit des Gesamtsystems auch ohne dedezierter mathematische Optimierung, durch manuell in den Szenarien gesetzten und dimensionierten Technologien, möglich.
- Der Entwurf als Thin-Client Modell vermeidet weitgehend intrinsische Berechnungen in der Komponente Wrapper selbst.

Die Spezifikation ermöglicht eine flexible Einbeziehung heterogener Softwarebausteine, sodass die einzelnen Partner die Konvertierungs-, Simulations- und Optimierungswerkzeuge weitgehend autark und parallel finalisieren konnten.

Fortschritte wurden sukzessive eingebunden und im Gesamtsystem evaluiert. Testinstanzen konnten horizontal und vertikal skaliert werden.

Kommunikationsschnittstellen

- User <-> Wrapper: Mittels Angabe von Verzeichnispfad der Eingabedaten (Workspace), Case und Simulationsstart- und Ende wird der Workflow gestartet. Als Ergebnis werden die MYNTS-Ausgaben interpretiert, sowie später durch den werusys Visualisierer dargestellt.
- WRAPPER <-> MYNTS (M-Bindings): Aufruf von MYNTS-Simulationsroutinen.
- WRAPPER <-> PyPSA (P-Bindings): Aufruf von PyPSA-Simulationsroutinen.
- WRAPPER <-> Optimierer: Start der Optimierungskomponente für die Erzeugung von Hypothesen.

Ablaufsteuerung

Netzdaten werden in der Key-Value-Form `attribute@object@time` zwischen den Komponenten durchgereicht:

- Attribut: z.B. *soc* := *State-of-Charge*.
- Objekt: z.B. *node0* := *Netzknoten 0*.
- Zeitstempel: z.B. 1541186477 für den 02.11.2018 20:21:17 Uhr :=UNIX-Zeitstempel (Integerwert).

Für die Wahrung der Kompatibilität werden SI-Einheiten verwendet. Für eine einfache Konvertierung werden im Wrapper selbst entsprechende Hilfsfunktionen bereitgestellt.

Speichertechnologien

Der Füllstand der Speichertechnologien (State-of-Charge =: SoC) wird direkt im Wrapper (für Mynts und PyPSA) abgelegt. Eine Konfigurationsdatei enthält eine Liste von entsprechenden Attributen und Objekten.

Für das Speicherverhalten werden sind die folgenden Strategien umgesetzt:

- Nutzung statischer Speicherprofile.
- Integration der Steuerung in das Gesamtoptimierungsproblem. Dies ist allerdings nicht uneingeschränkt möglich, da die einzelnen Softwarekomponenten autark rechnen.
- Regelgesteuerters Verhalten in Form von logischen Ausdrücken, die binäre Schalter steuern.

2.3 Softwarekonverter

(siehe Abschlussbericht der werusys GmbH)

2.4 Mynts

(siehe Abschlussbericht von Fraunhofer SCAI)

2.5 Optimierer

Hinsichtlich der Implementierung der Optimierungsverfahren ist eine Toolchain enstanden, die den gesamten Prozess von der Modellierung, bis zu der Bestimmung von konkreten wertmäßigen Ergebnissen begleitet. Abb. 2.5.1 zeigt die übliche Abfolge, die i.d.R. repetitiv ausgeführt wird, bis valide Ergebnisse vorliegen. Die Herausforderung liegt darin, die physikalischen Beschreibungen derart stark zu relaxieren, dass die Probleminstanzen auch bei sehr großen Modellen in endlicher (vertretbarer) Zeit lösbar bleiben.

Die Modellierung ist zunächst sehr approximativ gehalten, um die Berechnungen trotz NP-Schwere in endlicher Zeit durchzuführen.

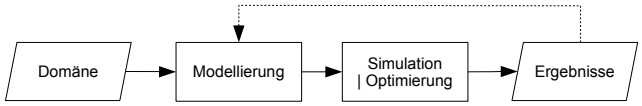

Abb. 2.5.1: Modellierung, Simulation, Optimierung

Zur softwareseitigen Trennung der Einzelaspekte wurde auch hier eine lose Kopplung von Softwarekomponenten vorgenommen. Diese Eigenentwicklungen waren vonnöten, um den hiesigen Anforderungen (Nachvollziehbarkeit der Ergebnisse, Behandlung von Problemen der Klasse MINLP) zu genügen.

Folgende Komponenten wurden spezifiziert und implementiert:

1. Modellierung mittels der domänenspezifischen Sprache (DSL) CEMPL.
2. Kompilation von CEMPL in die Zwischensprache IM.
3. Kompilation von IM in ein Optimierungproblem.
4. Ansteuerungskomponente für die Löser (Solver) BONMIN und Gurobi.
5. Begleitend dazu wurde eine Webumgebung (siehe Abbildung Abb. 2.5.2) auf- und ausgebaut.

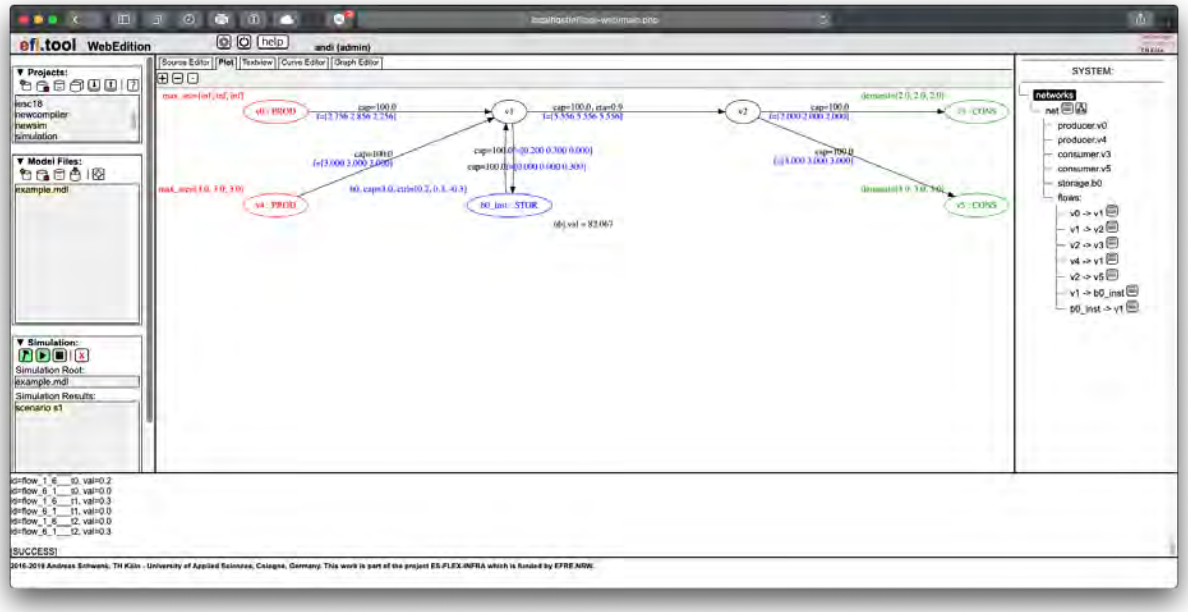

Abb. 2.5.2: Screenshot der webbasierten Testumgebung

Die vorliegende Komponentenunterteilung ermöglicht es, die einzelne Bausteine autark zu definieren, implementieren und zu verifizieren. Die Etablierung von formalen Schnittstellen ermöglicht es, menschenlesbare Konstrukte zu analysieren und sukzessive auszubauen und auszuwerten.

Auf alle Teilaspekte soll nun in diesem Kapitel näher eingegangen werden.

2.5.1 CEMPL

Die Lösung komplexer Problemstellungen aus Naturwissenschaften und Technik erfordert eine hinreichende Modellierung der realen Welt. Aus Gründen der Abstraktion sind zum Teil erhebliche Relaxierungen der exakten Beschreibung von physikalischen und ökonomischen Zusammenhänge vonnöten. Als oberstes Ziel ist die Wahrung der Systematik und Analysierbarkeit der Modelldaten zu gewährleisten, sodass diese generisch den unterschiedlichsten Weiterverarbeitungen (Berechnungen) genügen und in eine mathematische Ausdrucksform synthtisierbar sind.

Als Mittel der Wahl wurde entschieden, eine menschenlesebare Darstellungsweise in Form einer **domänenspezifischen Sprache** (Domain-Specific Language, DSL) für die Modellierung zu wählen. Die Definition der semantischen Zusammenhänge in eine *verständliche* und *nutzerfreundliche* Syntax fördert die interdisziplinäre Zusammenarbeit.

Die beiden Nutzungsszenarien der Sprache sind die folgenden:

1. Manuelle Definition von Modellen durch den DSL-Nutzer.
2. Maschinelle Erzeugung von Programmen in der Sprache CEMPL; z.B. durch Datenkonverter.

Sprachentwurf

Die Sprache „Cross Energy Management Programmming Language“ (CEMPL) gehört zu der Klasse der externen domänenspezifischen Sprachen. Damit obliegt sie nicht einer existierenden Hostsprache, und erlaubt die freie Definition der Syntax.

Der Aspekt der *Ausdrucksstärke* ist ein essentielles Ziel, sodass jegliche Instanzen (=Programme für die Beschreibung von Problemstellungen) möglichst kurz und übersichtlich gehalten werden können. Gleichzeitig wird damit das agile Konzept *Self-Documenting Code* erfüllt. Weiterhin wird die *Austauschbarkeit* von Daten gemäß eines *standardisierten* textuellen Formats gefördert.

Das unterliegende Programmierparadigma ist *deskriptiv* und *deklarativ*: Die Menge der Modellierungsdaten sind in einer beschreibende Form serialisiert. Auf die explizite Angabe von Algorithmen kann seitens des Nutzers verzichtet werden. So haben die Datensätze Potenzial möglichst umfassend und vielseitg ausgewertet zu werden. Spezifiziert wird stattdessen eine *Umgebung*, welche gewisse Freiheitsgrade offen lässt, die durch das System in Form von Entscheidungsvariablen selstständig gefunden werden. Die domänenspezifische Semantik, wie z.B. die Anwendung der Kirchhoff’schen Gesetze, wird implizit als *Ausführungsinformation* angenommen und kontextsensitiv während der Kompilation hinzugefügt.

Die Sprache enthält der Domäne der erneuerbaren Energien inhärente Schlüsselworte, sodass ein allseitiges Verständnis auch ohne Einarbeitung möglich ist. Die logischen Bestandteile der Sprache, also die die *Konzepte*, bestehen vor allem aus *Netzen*, *Zeitreihen*, *Speichertechnologien*, *Konvertertechnologien* und *Szenarien*.

Abb. 2.5.3 zeigt das *semantische Modell* in Form eines groben Klassendiagramms. Der Übersichtlichkeit wegen bleibt der Fokus auf dem reinen Datenmodell. Nicht alle Attribute werden gezeigt; auf die implementierte Sichtbarkeit (public, private) wird in der Darstellung ebenfalls keine Rücksicht genommen. Die Methoden für die Population durch den Parser und die betriebliche Nutzung durch den Codegenerator sind in der Übersicht weggelassen worden.

Das folgende Programm zeigt eine beispielhafte Verwendung.

```
energytype power:
        unit = kWh
producer a:
        profile = [10, 10, 10]
        co2 = 300 [g_kWh]
consumer b:
        profile = [5, 8, 3]
network net:
```

(Fortsetzung auf der nächsten Seite)

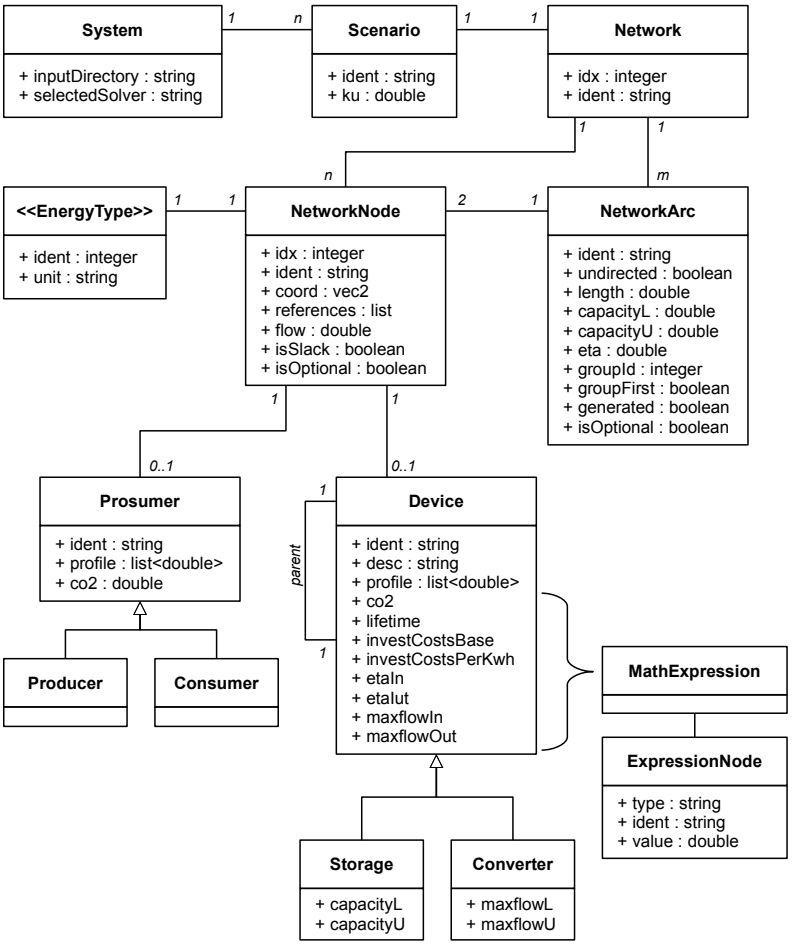

Abb. 2.5.3: Semantisches Modell der domänenspezifischen Sprache CEMPL

(Fortsetzung der vorherigen Seite)

```
        producer a: x= 0, y=0
        node     n: x= 5, y=0
        consumer b: x=10, y=0
        a -> n: capacity=100
        n -> b: capacity=100
scenario s1:
        use network net
```

Sprachevaluation

Im Folgenden wird eine Evaluation der Spracheigenschaften vorgenommen.

1. Eine *hohe Ausdrucksstärke* in Form von möglichst kurzen Programmen ist weitgehend erfüllt.
2. Die *Abdeckung* der Domäne ist beschränkt, sodass die Variation der synthetisierten Optimierungsaufgaben beherrschbar bleibt. Mindestens muss sichergestellt werden, dass auch sehr große Programmeingabeinstanzen lösbar bleiben.
3. CEMPL ist in sofern *sprachvollständig*, als dass möglichst wenige Ausführungsinformationen in der Sprache selbst undefiniert bleiben.
4. Die Sprache ist *modular* aufgebaut, sodass das Inkuldieren von Code möglich ist. Entsprechend ist paralleles Arbeiten durch verschiedene Stakeholder und eine Wartbarkeit gegeben.
5. Die *konkrete Syntax* zielt vor allem auf die Lesbarkeit des Codes ab. Schreibbarkeit und Erlernbarkeit durch den DSL-Nutzer stehen an zweiter Stelle.

Grammatik

Für die lexikalische und syntaktische Analyse wurde der *Parser-Generator* ANTLR (ANother Tool for Langauge Recognition) herangezogen. Auf Sprachwerkbänke wie z.B. Eclipse Xtext wurde verzichtet, um von der Leichtgewichtigkeit von ANTLR zu profitieren. Weiterhin ist die Implementierungssprache nicht auf Java beschränkt, sondern frei wählbar. Die im Folgenden dargestellten syntaktischen Regeln definieren die kontextfreie Sprache CEMPL. Bei der Definition musste durch Wahl eines Top-Down Parsers von Linksrekursion abgesehen werden. Da die Population des semantischen Modells über Listener erfolgt, wird hier kein Aktionscode angegeben.

```
grammar CEMPL;

cempl_program : ( stm )* EOF ;

stm : energytype | fileformat | profile | prosumer | device | network | scenario |␣
↪global_options ;

# -- options --

global_options : 'options' ':' BEGIN_OF_BLOCK (global_options_stm)* END_OF_BLOCK ;

global_options_stm : global_options_stm_solver ;

global_options_stm_solver : 'solver' '=' global_options_stm_solver_type NEWLINE ;

global_options_stm_solver_type : 'bonmin' | 'gurobi' ;

# -- energytype --
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
energytype_stm_unit : 'unit' '=' 'kWh' NEWLINE ;

energytype_stm : energytype_stm_unit ;

energytype : 'energytype' ID ':' BEGIN_OF_BLOCK (energytype_stm)* END_OF_BLOCK ;

# -- fileformat --

fileformat : 'fileformat' ID ':' BEGIN_OF_BLOCK (fileformat_stm)* END_OF_BLOCK ;

fileformat_stm : fileformat_stm_decimal | fileformat_stm_type | fileformat_stm_
↪formatstring | fileformat_stm_columns ;

fileformat_stm_decimal : 'decimal' '=' ( 'comma' | 'point' ) NEWLINE ;

fileformat_stm_type : 'type' '=' 'line' 'based' 'ascii' NEWLINE ;

fileformat_stm_formatstring : 'row' '=' '"' (fileformat_stm_formatstring_token)* '"'␣
↪NEWLINE ;

fileformat_stm_formatstring_token : '<DD>' | '<MM>' | '<YY>' | '<YYYY>' | '<hh>' | '
↪<mm>' | '<ss>'  /* date-time */ | '<ID>' | '<energytype>' | '<profile>' | '<real>'␣
↪| '<factor>' | '<co2>' | '/' | ':' | ';' | ',' ;

fileformat_stm_columns : 'column' 'labels' '=' ID ( ',' ID )* NEWLINE ;

# -- external profile (time series) --

profile : 'import' 'profile' 'as' ID 'from' 'file' PATH 'with' 'format' ID NEWLINE ;

# -- device --

device : device_type ID ':' BEGIN_OF_BLOCK (device_stm)* END_OF_BLOCK ;

device_type : 'storage' | 'converter' ;

device_stm : device_description | device_parent | device_io | device_investment_costs␣
↪| device_profile | storage_energytype | storage_capacity | converter_max_flow ;

device_description : ('description' | 'desc') '=' STR NEWLINE ;

device_parent : 'parent' '=' ID NEWLINE ;

converter_max_flow : 'max_flow' '=' const_range ( 'kWh' )? NEWLINE ;

storage_energytype : 'energytype' '=' ID NEWLINE ;

device_io : device_io_direction ':' BEGIN_OF_BLOCK ( device_io_stm )* END_OF_BLOCK ;

device_io_direction : 'input' | 'output' ;

device_io_stm : io_eta | storage_max_flow | converter_energytype ;

io_eta : 'eta' '=' const_expression NEWLINE ;

storage_max_flow : 'max_flow' '=' const_expression ( 'kWh' )? NEWLINE ;
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
converter_energytype : 'energytype' '=' ID NEWLINE ;

storage_capacity : 'capacity' '=' const_range ( 'kWh' )? NEWLINE ;

device_investment_costs : 'investment_costs' ':' BEGIN_OF_BLOCK ( device_investment_
↪costs_stm )* END_OF_BLOCK ;

device_investment_costs_stm : device_investment_costs_stm_base | device_investment_
↪costs_stm_per_kwh ;

device_investment_costs_stm_base : 'base' '=' const_expression ( 'EUR' )? NEWLINE ;

device_investment_costs_stm_per_kwh : 'per_kWh' '=' const_expression ( 'EUR' )?␣
↪NEWLINE ;

device_profile : 'profile' '=' float_list NEWLINE ;

# -- network --

network : 'network' ID ':' BEGIN_OF_BLOCK (network_stm)* END_OF_BLOCK ;

network_stm : network_stm_add_nodes | network_stm_add_edges ;

network_stm_add_nodes_type : 'producer' | 'consumer' | 'storage' | 'node' | 'slack' ;

network_stm_add_nodes_optional : ( 'optional' )? ;

network_stm_add_nodes_additional_types : ( 'or' ID )* ;

network_stm_add_nodes : network_stm_add_nodes_optional network_stm_add_nodes_type ID␣
↪network_stm_add_nodes_additional_types ( 'as' ID )? ':' ( network_stm_add_nodes_stm␣
↪(',' network_stm_add_nodes_stm )* )? NEWLINE ;

network_stm_add_nodes_stm : network_stm_add_nodes_stm_pos_x | network_stm_add_nodes_
↪stm_pos_y ;

network_stm_add_nodes_stm_pos_x : 'x' '=' constant ;

network_stm_add_nodes_stm_pos_y : 'y' '=' constant ;

network_stm_add_edges_optional : ( 'optional' )? ;

network_stm_add_edges_additional_types : ( 'or' ID )* ;

network_stm_add_edges_type : 'line' | 'converter' ID network_stm_add_edges_additional_
↪types ;

network_stm_add_edges_dest_list : (',' ID)* ;

network_stm_add_edges_alias : ( 'as' ID )? ;

network_stm_add_edges : network_stm_add_edges_optional network_stm_add_edges_type␣
↪network_stm_add_edges_alias ':' ID network_stm_add_edges_stm_connect_direction ID␣
↪network_stm_add_edges_dest_list ( ':' network_stm_add_edges_stm (',' network_stm_
↪add_edges_stm )* )? NEWLINE ;

network_stm_add_edges_stm_connect_direction : '->' | '<->' ;
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
network_stm_add_edges_stm : network_stm_add_edges_stm_capacity | network_stm_add_
↪edges_stm_eta ;

network_stm_add_edges_stm_capacity : 'capacity' '=' const_expression ;

network_stm_add_edges_stm_eta : 'eta' '=' const_expression ;

# -- prosumers (producers and consumers) --

prosumer : prosumer_type ID ( prosumer_array )? ':' BEGIN_OF_BLOCK (prosumer_stm)*␣
↪END_OF_BLOCK ;

prosumer_type : 'producer' | 'consumer' ;

prosumer_array : '[' ']' ;

prosumer_stm : fileimport | prosumer_energytype | prosumer_co2 | prosumer_output |␣
↪prosumer_profile | prosumer_description ;

prosumer_description : ('description' | 'desc') '=' STR NEWLINE ;

prosumer_energytype : 'energytype' '=' ID NEWLINE ;

prosumer_co2 : 'co2' '=' const_expression ( 'g_kWh' )? NEWLINE ;

prosumer_output : 'output' '=' const_expression ( 'kWh' )? NEWLINE ;

prosumer_profile : 'profile' '=' float_list NEWLINE ;

# -- scenarios --

scenario : 'scenario' ID ':' BEGIN_OF_BLOCK (scenario_stm)* END_OF_BLOCK ;

scenario_stm : scenario_stm_use_network | scenario_stm_upper_bound_costs ;

scenario_stm_upper_bound_costs : 'limit' 'costs' 'to' const_expression ( 'EUR' )?␣
↪NEWLINE ;

scenario_stm_use_network : 'use' 'network' ID NEWLINE ;

# -- file imports --

fileimport : 'file' ':' fileimport_path 'format' '=' ID ( fileimport_lines )? ( ':'␣
↪BEGIN_OF_BLOCK (fileimport_stm)* END_OF_BLOCK )? ;

fileimport_path : PATH ;

fileimport_lines : 'lines' '=' INTEGER '..' INTEGER ;

fileimport_stm : fileimport_stm_datacolumns ;

fileimport_stm_datacolumns : 'data' 'columns' ':' ID ( ',' ID )* ;

# -- complex data types --

float_list_entry : const_expression ;
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
float_list : '[' (float_list_entry (',' float_list_entry)*)? ']' ;

tuple2 : '(' const_expression ',' const_expression ')' ;

const_range : const_expression | const_expression const_range_dots const_expression ;

const_range_dots : '..' ;

const_expression : const_expression_add ;

const_expression_add : const_expression_add const_expression_add_sym const_expression_
↪mul | const_expression_mul ;

const_expression_mul : const_expression_mul const_expression_mul_sym const_expression_
↪pow | const_expression_pow ;

const_expression_pow : const_expression_pow const_expression_pow_sym const_expression_
↪unary | const_expression_unary ;

const_expression_unary : const_expression_unary_constant | '(' const_expression_add ')
↪' ;

const_expression_unary_constant : constant ;

const_expression_mul_sym : '*' | '/' ;

const_expression_add_sym : '+' | '-' ;

const_expression_pow_sym : '^' ;

expression : expression_relation ;

expression_relation : expression_relation expression_relation_sym expression_add |␣
↪expression_add ;

expression_relation_sym : '>' | '<' | '>=' | '<=' | '=' ;
expression_add : expression_add expression_add_sym expression_mul | expression_mul ;

expression_mul : expression_mul expression_mul_sym expression_pow | expression_pow ;

expression_pow : expression_pow expression_pow_sym expression_unary | expression_
↪unary ;

expression_unary : constant | expression_function_call | expression_variable | '('␣
↪expression_add ')' ;

expression_function_call : expression_function_call_name '(' expression_add ')' ;

expression_function_call_name : 'exp' | 'sin' | 'cos' ;

expression_variable : expression_identifier ( '.' expression_identifier )* ;

expression_identifier : 'capacity' | 'investment_costs' | ID ;

expression_mul_sym : '*' | '/' ;

expression_add_sym : '+' | '-' ;
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
expression_pow_sym : '^' ;

constant : ('-')? REAL | ('-')? INTEGER ;

# -- terminals --

REAL : 'infinity' | 'inf' | DIGIT0 '.' DIGIT0* INTEGER_SUFFIX? | DIGIT DIGIT0* '.'␣
↪DIGIT0* INTEGER_SUFFIX? ;

INTEGER : DIGIT DIGIT0* INTEGER_SUFFIX? | DIGIT0 INTEGER_SUFFIX? ;

fragment INTEGER_SUFFIX : 'n' | 'u' | 'm' | 'k' | 'M' | 'G' | 'T' ;

fragment DIGIT0 : '0' .. '9' ;

fragment DIGIT : '1' .. '9' ;

ID : ID_START ID_CHAR* ;

fragment ID_START : ('a' .. 'z') | ('A' .. 'Z') | '_' ;

fragment ID_CHAR : ID_START | ('0' .. '9') ;

PATTERN : ',' | ':' | ';' | '/' | '.' ;

PATH : '"' ID_START ID_CHAR* ( '.' ID_START ID_CHAR* )? '"' ;

STR : '\'' (~['"] )* '\'' ;

BEGIN_OF_BLOCK : '@0' '\n' '@+' ;

END_OF_BLOCK : '@-' ( '@0' )? ;

NEWLINE : '@0' ( ( '\t' | '\n' )* '@0' )* ;

WS : [ \t\r\n]+ -> skip ;

COMMENT : '#' ~( '\r' | '\n' )* -> skip ;
```

2.5.2 Codegenerierung

Sie Definition der Sprache CEMPL ermöglicht syntaktische gesehen vielfältige Beschreibungen. Die *Zielsprache* ist ein formales mathematisches Optimierungsproblem. In CEMPL geschriebene Programmer werden *kompiliert* (nicht interpretiert). Die Codegenerierung wird in zwei konsekutiven Schritten durch dedizierte Compiler durchgeführt.

1. **Compiler A**: In einem ersten *Kompilationsschritt* werden in CEMPL geschriebene Programme in eine vereinfachte Darstellung übersetzt. Der Ausgabecode ist *modellbewusst*, kann also noch auf das semantische Modell zurückgeführt werden. Diese Zwischensprache wird hier *„Intermediate Language“, kurz IM* genannt und wird im nächsten Unterkapitel beschrieben.

2. **Compiler B**: In einem zweiten Schritt wird die Zwischensprache in ein *mathematisches Optimerungsprobem* in Form einer Zielfunktion und eine Menge von Nebenbedingungen kompiliert. Hierzu wurde ein einfaches, menschenlesbares trennzeichengesteuertes Format definiert, welches *modellignorant* ist. Dieses ist unten im Abschnitt *Optimization Language (OL)* beschrieben und begünstigt die Wartbarkeit (vgl. mit dem oft eingesetzen Quasistandard NL).

Die eigens für das Projekt geschriebene Softwarekomponente **Solver** interpretiert schließlich das mathematische Optimierungsproblem und gibt die Besetzung der kontinuierlichen und diskreten Entscheidungsvariablen aus.

2.5.3 Intermediate Language (IM)

Da bisher im Rahmen der Sektorenkopplung keine standardisierte Beschreibungsform für die Datenhaltung von Optimierungsproblemen existiert, wird im Folgenden eine Neudefinition vorgenommen. Diese ist essentiell, um den Debuggingprozess zu begünstigen. Die Zwischensprache bildet vor allem generische Ausdrücke auf konkrete Instanzmengen ab („Ausrollen“).

Eingabe: CEMPLE

Ausgabe: IM

Die IL selbst kann mittels Übersetzer in ein mathematisches Optimierungsproblem im Format OL (siehe nächster Abschnitt) überführt werden:

$$\min_x f_0(x) \;\; s.t. \; f_i(x) \leq 0 \;\; \forall i \in [m]$$

Mit $x \in \mathbb{R}^n$ und $x_j \in \mathbb{Z}$ für einige $j \in [m]$.

Programme in IM sind in die vier Dateien `global.txt`, `graph.txt`, `dev.txt`, `st.txt` gegliedert. Diese werden nun spezifiziert.

Die folgenden Datentypen finden Verwendung:

- `BOOL`: Boolscher Datentyp (default: `false`).
- `INT`: Ganzzahliger Datentyp (default: `0`).
- `REAL`: Reellwertiger Datentyp (default: `0.0`).
- `ID`: Bezeichner.

Globale Variablen (`global.txt`)

Auflistung globaler Konstanten mit Gültigkeit für den gesamten Berechnungslauf.

Attribute:

- `ku`: Obere Schranke der Summe aller Investitionskosten in EUR
- `gu`: Obere Schranke der Summe aller Treibausgase in g/kWh

Grammatik:

```
root = line*;
line = (cup | gup), "\n";
cup = "ku", "=", REAL;
gup = "gu", "=", REAL;
```

Bespiel:

```
ku=100000
```

Netzstruktur (`graph.txt`)

Graphische Beschreibung des Netzes. Dabei wird erst die Menge der Knoten und dann die Menge der Kanten beschrieben.

Attribute:

- `v`: Knoten.
- `e`: Kante.
- `x`: kartesische x-Koordinate.
- `y`: kartesische y-Koordinate.
- `c`: Leitungskapazit (obere Flussgrenze) in kWh.
- `eta`: Effizienzfaktor [0,1].
- `opt`: Instanz ist optional; diskrete Entscheidungsvariable.
- `slack`: Slack.

Grammatik:

```
root = v_list, "%%\n", e_list;
v_list     = v*;
v          = ID, (",", v_attr)*, "\n";
v_attr     = v_x | v_y | v_slack | v_opt;
v_x        = "x", "=", REAL;
v_y        = "y", "=", REAL;
v_slack    = "slack", "=", ("true" | "false");
v_opt      = "opt", "=", BOOL;
e_list     = e*;
e          = ID, ",",              # edge ID
             ID, ",",              # ID of the start-vertex
             ID, ",",              # ID of the end-vertex
             (",", e_attr)*, "\n";
e_attr     = e_c | e_eta | e_opt;
e_c        = "c", "=", REAL;
e_eta      = "eta", "=", REAL;
e_opt      = "opt", "=", BOOL;
```

Bespiel:

```
p1, x=0.0, y=0.0,
n1, x=5.0, y=0.0, slack=true
c1, x=10.0, y=0.0, opt=true
%%
e0, p1, n1, c=100.0, eta=0.9
e1, n1, c1, c=100.0
```

Technologien (`dev.txt`)

Eine Technologiedatenbank besteht aus drei Teilen.

1. Liste der Technologieinstanzen
2. Vertexbasierte Attribute
3. Kantenbasierte Attribute

Attribute:

- `v`: Knoten.
- `e`: Kante.
- `cap_l`: Untere Schranke für die Speicherkapazität in kWh.
- `cap_u`: Obere Schranke für die Speicherkapazität in kWh.
- `num`: Anzahl der verfügbaren Technologien (kombinatorische Optimierung).
- `idx`: Index der Knoten/Kantendeklaration.
- `dev`: Referenz zur Technologie
- `k0`: Fixe Investitionskosten in EUR.
- `k1`: Variable Investitionskosten (basierend auf der tatsächlichen Kapazität) in EUR.
- `k`: Gesamtkosten $k := k0 + k1 \cdot cap$.

Grammatik:

```
root = d_list, "%%\n", v_list, "%%\n", e_list;
d_list     = d*;
d          = ID, (",", d_attr)*, "\n";
d_attr     = d_type | d_num | opt;
d_type     = "storage" | "converter";
d_num      = INT;
d_opt      = "True" | "False";
v_list     = v*;
v          = ID, (",", v_attr)*, "\n";
v_attr     =    v_dev | v_idx | v_typename | v_cap_l |
              | v_cap_u | k0 | k1;
v_dev      = ID;
v_idx      = INT;
v_typename = ID;
v_cap_l    = "c_l", "=", REAL;
v_cap_u    = "c_u", "=", REAL;
e_list     = e*;
e          = ID, (",", e_attr)*, "\n";
e_attr     =    e_dev | e_idx | e_cap_l | e_cap_u | e_eta
             | k0 | k1;
v_dev      = ID;
v_idx      = INT;
e_cap_l    = "c_l", "=", REAL;
e_cap_u    = "c", "=", REAL;
e_eta      = "eta", "=", REAL;
k0         = "k0", "=", REAL;
k1         = "k1", "=", REAL;
```

Bespiel:

```
s, type=storage, num=1, opt=False
%%
s, dev=s, idx=0, typename=s, c_l=0.5, c_u=3, k0=100, k1=20
%%
e3, dev=s, idx=0, c=0.3, eta=0.81
e4, dev=s, idx=0, c=0.3, eta=0.82
```

Quellen und Senken (`st.txt`)

Attribute:

- `p`: Erzeuger (producer).
- `c`: Verbraucher (consumer).
- `v`: Wert(e) (value(s)) in kWh.
- `co2`: Emissionen in g/kWh.

Grammatik:

```
root = p_list, "%%\n", c_list;
p_list    = p*;
p         = ID, (",", p_attr)*, "\n";
p_attr    = p_v | p_co2;
p_v       = "v", "=", REAL_LIST;
p_co2     = "co2", "=", REAL;
c_list    = c*;
c         = ID, (",", c_attr)*, "\n";
c_attr    = c_v;
c_v       = "v", "=", REAL_LIST;
REAL_LIST = "[", REAL, (",", REAL)*, "]";
```

Bespiel:

```
p1, v=[1.0 1.0 1.0 0.0 0.0], co2=100.0
%%
c1, v=[0.0 0.0 1.0 1.0 0.0]
```

Vollständiges Beispiel

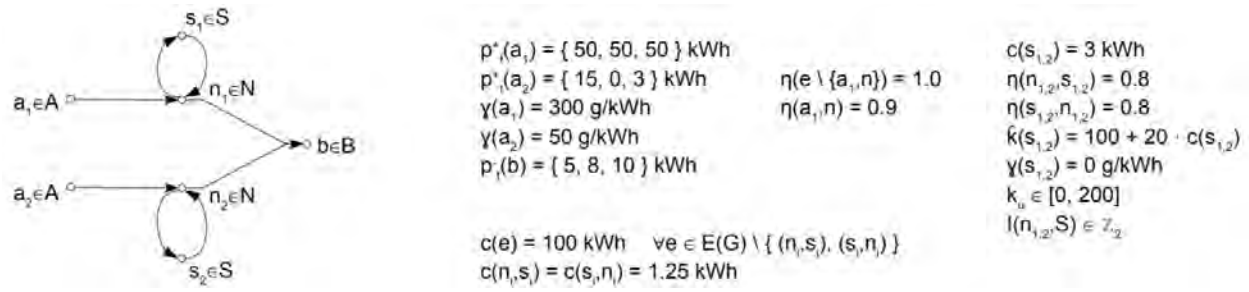

Abb. 2.5.4: Beispiel: Intermediate Language

global.txt:

```
ku=200
```

graph.txt:

```
a1, x=0, y=0
a2, x=0, y=5
n1, x=5, y=0
n2, x=5, y=5
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
b, x=10, y=2.5
s1, x=5, y=-5, opt=true
s2, x=5, y=10, opt=true
%%
e0, a1, n1, c=100, eta=0.9
e1, a2, n2, c=100
e2, n1, b, c=100
e3, n2, b, c=100
e4, n1, s1, c=inf
e5, s1, n1, c=inf
e6, n2, s2, c=inf
e7, s2, n2, c=inf
```

dev.txt:

```
s1, type=storage, num=1, opt=True
s2, type=storage, num=1, opt=True
%%
s1, dev=s1, idx=0, typename=s, c_l=3, c_u=3, k0=100, k1=20
s2, dev=s2, idx=0, typename=s, c_l=3, c_u=3, k0=100, k1=20
%%
e4, dev=s1, idx=0, c=1.25, eta=0.8
e5, dev=s1, idx=0, c=1.25, eta=0.8
e6, dev=s2, idx=0, c=1.25, eta=0.8
e7, dev=s2, idx=0, c=1.25, eta=0.8
```

st.txt:

```
a1, v=[50 50 50], co2=300
a2, v=[15 0 3], co2=50
%%
b, v=[5 8 10]
```

2.5.4 Optimization Language (OL)

Um die Ansteuerung der Löser zu entkoppeln, wurde das neue Format OL (Optimization Language) definiert. Im Gegensatz zu z.B. weit verbreiteten NL-Formats sind die Programme menschenlesbar. Dies ist unabdingbar, um den Systemausbau wartbar zu halten.

Beispielprogramm

```
real, 0 : x : 100, 0  # kontinuierliche Entscheidungsvariable
int, 0 : y : 100, 0   # diskrete Entscheidungsvariable
%%
min x*y               # Zielfunktion
%%
c0, 5 : x + y : 10    # Nebenbedingung
```

Die Syntax ist per Prozentzeichen in die folgenden drei getrennten Blöcke unterteilt:

1. Definition der *Entscheidungsvariablen:* Typ, untere Grenze, Bezeichner, obere Grenze, Startwert.
2. *Zielfunktion:* Minimierung oder Maximierung.
3. *Nebenbedingungen:* Bezeichner, untere Grenze, Mathematischer Ausdruck, obere Grenze.

Sowohl die Zielfunktion, als auch die Nebenbedingungen können beliebige mathematische Terme in Infixnotation beinhalten. Die Menge der unterstützten einstelligen Funktionen ist sin, cos, exp, log usw.

2.5.5 Solver

Die für dieses Projekt geschriebene Softwarekomponente *Solver* löst ein in der Sprache *Optimization Language (OL)* definiertes mathematisches Optimierungsproblem und schreibt die wertmäßigen Ergebnisse (Skalare) in eine Textdatei. Die Komponente ist zwei Softwareprojekte gegliedert, die im Folgenden beschrieben werden.

OPT-Wrapper

Eine in Pyhton definierte Wrapperklasse mit Parsing- und Ausgbabemethoden für das OL-Format. Der Kern liegt in der *Vereinfachung* von mathematischen Optimierungsproblemen. *Compiler B* erzeugt zwar semantisch korrekten Code; legt aber der Einfachheit und Wartbarkeit wegen, keinen Fokus auf eine prägnante Formulierung. Entsprechend ist die Anzahl an Entscheidungsvariablen sehr hoch. Eigentlich lineare Terme sind meist durch Klammerung so formuiert, dass ein Solver ggf. von Nichtlinearität ausgeht. Zum Beispiel werden die folgenden Schritte vorgenommen:

1. Entscheidungsvariablen für die gilt, dass die obere und untere Grenze numerisch gleich sind, können in allen Nebenbedingungen durch Konstanten ersetzt werden.
2. Die Grenzen der Nebenbedingungen, deren mathematischer Audruck lediglich aus einer Entscheidungsvariablen besteht, können elimiert werden, wenn die Grenzen in die Definition der Entscheidungsvariablen selbst übernommen werden.
3. usw.

OPT-Solver

Der Part der Solverintegration musste manuell bewerkstelligt werden, da die sonst üblicherweise eingesetzten Einbettungsubgebungen proprietär sind und hohe Lizenzgebühren nach sich ziehen (z.B. AMPL). Weiterhin sollte die Kompatibilität zu dem in MYNTS eingesetzten NLP-Solver IPOPT auf numerischer Ebene gewährleistet bleiben.

Der OPT-Solver ist ein für dieses Projekt in der Programmiersprache C++ geschriebener Löser, der den freien MINLP Solver BONMIN als Bibliothek einbindet. Die Wahl von BOMIN soll den Einsatz von freier Software für Probleme aus dem Energiebereich fördern. Auch die Softwarekomponente OPT-Solver liest zunächst das im OL-Format spezifizierte mathematische Optimierungspoblem ein. Die Ansteuerung von BONMIN geschieht über die Ausimplementierung des Interfaces *TMINLP*. Da BONMIN Eingaben auf Low-Level Ebene erwartet, liegen die Herausforderungen im symbolischen Management der mathematischen Terme. Dazu gehört insbesondere der Support der *automatischen Ableitung*. Durch die Verwendung der Programmiersprache (interpretierte Sprachen wie Python scheiden aus Performanzgründen aus) C++ ist für die internen Datenstrukturen ein aufwändiges, manuelles Speichermangement erforderlich.

Aus Sicht der Kodierung ist eine aufwändige Auswertung des Eingabeproblemens vonnöten (ähnlich wie für den nichtlinearen Solver IPOPT). Die folgenden Punkte müssen per Software extrahiert, definiert und gepflegt werden:

1. Bestimmen der Variablentypen: Binär, Integer, kontinuierlich.
2. (Rekursives) Bestimmen der Linearität der Entscheidungsvariablen: werden diese ausschlielich linear, oder auch in einem nichtlinearen Kontext in den Nebenbedingungen eingesetzt?
3. (Rekursives) Bestimmen der Linearität der Nebenbendingungen selbst.
4. Bestimmen der folgenden Parameter: Anzahl der Entscheidungsvariablen, Anzahl der Nebenbedingungen, Anzahl der Elemente der *jakobischen* Matrix die nicht Null sind, Anzahl der element der *hessischen* Matrix die nicht Null sind.
5. Bestimmen der unteren und oberen Grenze, sowie der numersichen Startwerte aller Entscheidungsvariablen. Letztere sind Bestandteil des OL Formates (s.o.).

6. Callback für die Berechnung der folgenden Funktionen unter Berücksichtigung der aktuellen Besetzung der Entscheidungsvariablen: *Zielfunktion*, *partielle Ableitungen der Zielfunkion*, *Nebenbedingungen*, *jacobischen Matrix*, *hessischen Matrix*.
7. Callback für die Rückgabe der endgültigen Lösung.

BONMIN liefet exakte Ergebnisse, wenn ausschließlich konvexe Funktionen für die Zielfunktion und für die Nebenbedingungen verwendet werden. Andernfalls werden die Entscheidungsvarialen nur approximativ bestimmt. Die Wahl des konkreten Algorithmus (B-BB, B-QO, B-QG oder B-Hyp) für die Lösung hängt maßgeblich vom Eingabeproblem ab.

2.5.6 Weboberfläche der Algorithmentestplattform

Für einen eingängigen Bedienungsworkflow der Optimerungssalgorithmik wurde eine Webumgebung geschaffen. In den Screenshots, Abb. 2.5.5, Abb. 2.5.6 und Abb. 2.5.7 sieht man exemplarische Auszüge der Funktionalität.

Der Webservice besteht aus kleinen und agilen Services. Dazu wurde sich an den folgenden Sprachen bedient.

1. *HTML und CSS*: Frontendumsetzung.
2. *JavaScript*: Kommunikation mit dem Server.
3. *PHP*: Start- und Rückmeldekommunikation von Diensten; z.B. Kompilation.
4. *Bash-Skripte* und *Python-Skripte*: Betriebssystemnahe Ablaufsteuerung.

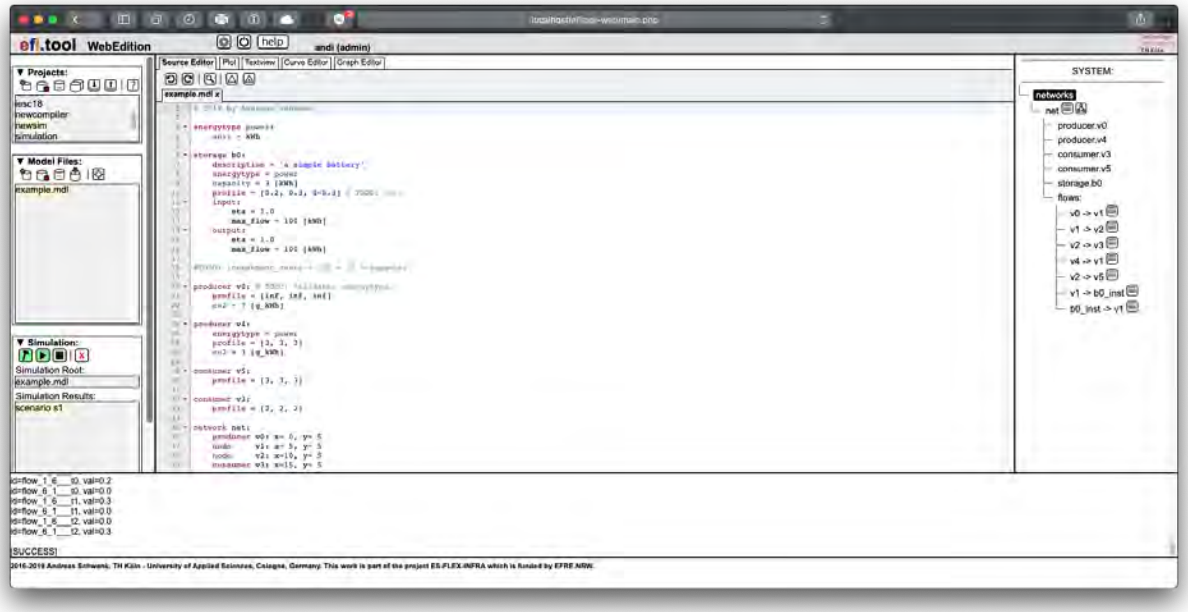

Abb. 2.5.5: Screenshot Algorithmentestplattform 1

Die folgenden Features wurden realisiert:

- Dateimanagement der Modelldaten.
- Eine Integrierte Entwicklungsumgebung (IDE) mit Editoren für die Sprache CEMPL. Dazu gehört z.B. das Hervorheben von Schlüsselwörter (Syntax Highlighting),
- Steuerung des Kompilierungsablaufs (Compiler A, Compiler B) mit Ausgabe von Fehlermeldungen. Das Gesamtsystem wird hierarchisch in einer Baumstruktur dargestellt.

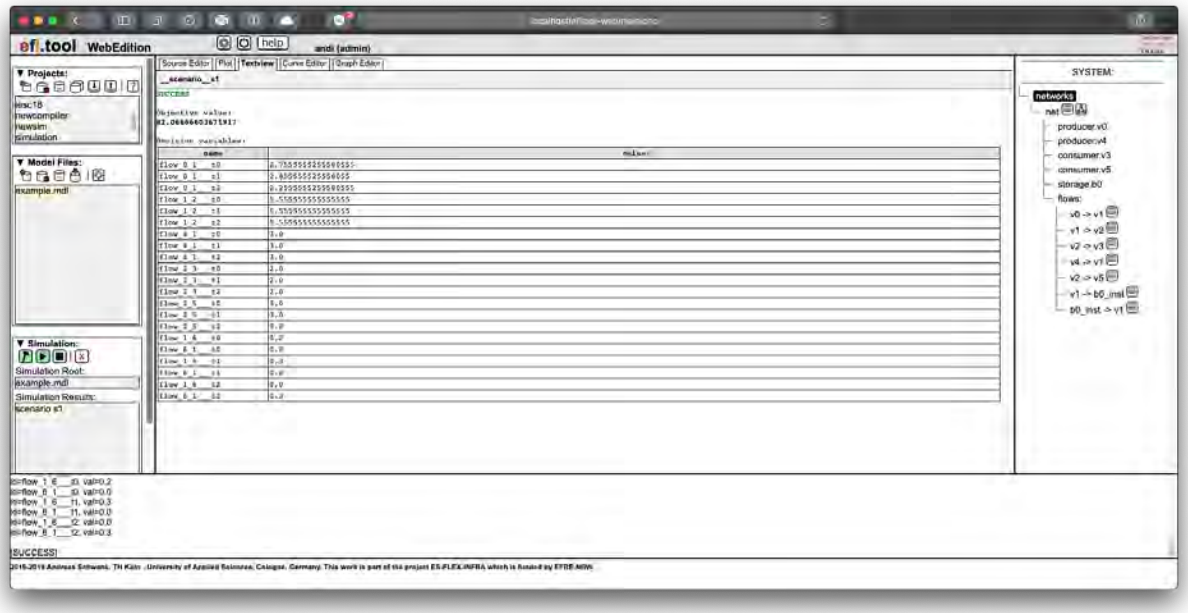

Abb. 2.5.6: Screenshot Algorithmentestplattform 2

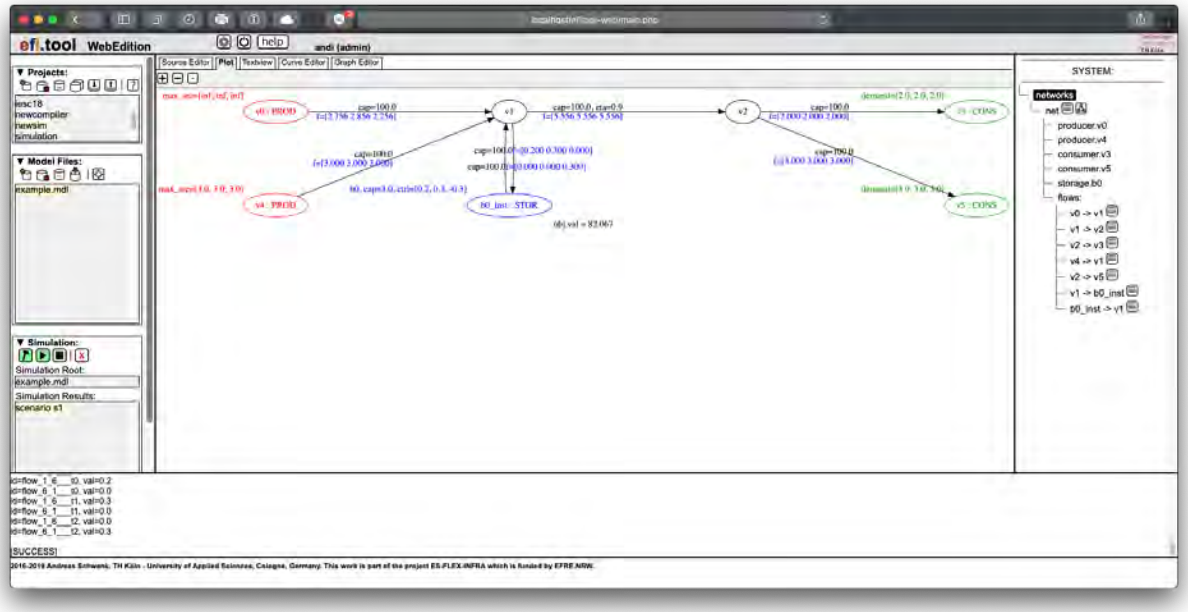

Abb. 2.5.7: Screenshot Algorithmentestplattform 3

- Start des Berechnungsablaufs (Ausführung des Solvers).
- Anzeigen der Ergebnisse in Form von Tabellen und Plots.
- Die im Rahmen der Modellierung anfallenden Kennlinien aus Datenblättern können mittels eines Kurveneditors nachgezeichnet und dann approximiert werden.

2.6 Weboberfläche

(siehe Abschlussbericht der werusys GmbH)