Simulation von Schaltnetzminimierungen

2017

Katharina Melisande Albrecht
Heinrich-Hertz-Gymnasium

Jonas Wanke
Heinrich-Hertz-Gymnasium

Inhalt

1. Abkürzungsverzeichnis

BCD	*Binary Coded Decimal* – Binäre Darstellung einer einzelnen Ziffer mit vier Bits
KDNF	*Kanonisch Disjunktive NormalForm* – siehe Abschnitt 3.4.1
SOP	*Sum Of Products* – Disjunktive Normalform, siehe Abschnitt 3.4
UI	*User Interface* - Benutzeroberfläche
WPF	*Windows Presentation Foundation* – Grafikbibliothek von Windows

2. Kurzfassung

Jeder, der sich schon einmal genauer mit logischen Schaltungen auseinander gesetzt hat, sollte wissen, dass diese schnell unübersichtlich werden. Setzt man sie zudem in ein reales Schaltnetz um, sind diese Schaltungen oft ineffizient und ausschweifend und könnten meist durch einfachere ersetzt werden. Doch wie gelangt man zu einer solchen verkleinerten Schaltung? Sehr leicht gelingt dies durch Anwendung einer Schaltnetzminimierung wie beispielsweise der Quine-McCluskey-Minimierung, welche eine minimale (optimal minimierte) logische Schaltung ausgibt. Derartige Algorithmen haben allerdings den Nachteil oft schwer verständlich zu sein. Darum erstellt dieses Projekt ein Programm, welches Quine-McCluskey-Minimierungen ausführt und das Verfahren dahinter grafisch aufbereitet. Es ist besonders zur Veranschaulichung geeignet, wenn die Minimierung im Rahmen des Informatikunterrichts an Schulen behandelt wird, da solche Mittel bislang fehlen.

3. Begriffsklärungen

3.1. Literal

Ein Literal ist in der Booleschen Algebra eine nichtnegierte oder eine negierte Variable.

Beispiel: x_1 und $\overline{x_1}$ sind Literale der Variable x_1 .

3.2. Konjunktion

Eine Konjunktion ist in der Logik eine Verknüpfung von mehreren Aussagen, die genau dann wahr ist, wenn alle verknüpften Aussagen wahr sind. In der Booleschen Algebra sind diese Aussagen mit Literalen oder komplexeren booleschen Funktionen gleichzusetzen.

Beispiel: $x_1\overline{x_2}$, $x_1 \cdot \overline{x_2}$, $x_1 \wedge \overline{x_2}$ sind jeweils Konjunktionen zweier Literale: x_1 und $\overline{x_2}$. Es müssen aber nicht alle Variablen vorkommen: $\overline{x_1}x_3$

Hinweis: Alle drei Schreibweisen sind identisch, im folgenden wird jedoch die erste verwendet.

3.2.1. Konjunktionsterm

Eine Konjunktion ist genau dann ein Konjunktionsterm, wenn alle vorkommenden Aussagen Literale sind.

Beispiel: $x_1\overline{x_2}$ ist ein Konjunktionsterm, $x_1(x_2 + \overline{x_3})$ jedoch nicht.

3.2.2. Vollkonjunktion / Minterm

Eine Konjunktionsterm ist genau dann ein Minterm (eine Vollkonjunktion), wenn alle *n* Variablen einer *n*-stelligen booleschen Funktion darin vorkommen.

Beispiel: $x_1\overline{x_2}x_3$, wenn es in der Funktion genau drei Variablen gibt.

3.3. Disjunktion

Eine Disjunktion ist in der Logik eine Verknüpfung von mehreren Aussagen, die genau dann wahr ist, wenn mindestens eine der verknüpften Aussagen wahr ist. In der Booleschen Algebra sind diese Aussagen mit Literalen oder komplexeren booleschen Funktionen gleichzusetzen.

Beispiel: $x_1 + \overline{x_2}$, $x_1 \vee \overline{x_2}$ sind jeweils Disjunktionen zweier Literale: x_1 und $\overline{x_2}$.

Hinweis: Die beiden Schreibweisen sind identisch, im folgenden wird jedoch die erste verwendet.

3.4. Disjunktive Normalform (DNF)

Eine logische Funktion aus disjunktiv verknüpften Konjunktionstermen.

Beispiel: $x_1 x_2 x_3 + x_1 \overline{x_2}$

3.4.1. Kanonisch disjunktive Normalform (KDNF)

Eine disjunktive Normalform ist genau dann kanonisch, wenn alle vorkommenden Konjunktionsterme Vollkonjunktionen/Minterme und voneinander verschieden sind.

Beispiel: $x_1 x_2 x_3 + x_1 \overline{x_2} x_3 + x_1 \overline{x_2}\,\overline{x_3}$ ist die KDNF des Beispiel der disjunktiven Normalform.

3.5. Primterm / Primimplikant

Ein Term einer Disjunktiven Normalform, welcher nicht weiter vereinfacht werden kann.

Beispiel: In der Funktion $x_1 x_2 + x_1 \overline{x_2} + \overline{x_2}$ ist $\overline{x_2}$ ein Primterm, da man ihn nicht weiter vereinfachen kann. $x_1 x_2 + x_1 \overline{x_2}$ hingegen kann man zu x_1 zusammenfassen.

3.5.1. Kernprimimplikant / Kernimplikant

Ein Primterm, der bei der Darstellung einer Funktion nicht überflüssig ist.

4. Quine-McCluskey-Minimierung

Die Quine-McCluskey-Minimierung benutzt die kanonisch disjunktive Normalform, um ausgehend von dem Prinzip, eine Variable eliminieren zu können, sollte sie in zwei Mintermen einmal nichtnegiert und einmal negiert vorkommen, eine Minimierung der Ausgangsfunktion zu bewirken:

$$x_1 x_2 + x_1 \overline{x_2} = x_1 (x_2 + \overline{x_2}) = x_1$$

Minimiert bedeutet in diesem Fall, dass die Gesamtzahl der Konjunktionen und Disjunktionen minimal ist. Nachdem die kanonisch disjunktive Normalform erstellt wurde, werden die enthaltenen Minterme in binärer Schreibweise in einer Tabelle aufgetragen. 0 bedeutet, dass die Variable an der Stelle eine Negation, 1, dass sie keine Negation aufweist. Sortiert werden die Terme dabei aufsteigend nach Anzahl der Einsen, wobei die verschiedenen Anzahlen jeweils durch eine horizontale Linie abgetrennt werden. So entstehen mehrere Blöcke.

Anschließend wird von je zwei übereinanderliegenden Blöcken jeder Term aus dem oberen mit jedem Term aus dem unteren Block verglichen. Falls sich diese nur durch Negation einer Variablen an einer Stelle unterscheiden, wird diese Variable eliminiert (gekennzeichnet durch _), und der neue Term in die nächste Spalte (und dort wieder in den passenden Block, abhängig von der Anzahl der Einsen) eingetragen. Danach werden alle dadurch abgedeckten (verwendeten) Terme durch ein Häkchen markiert. Dieses Prinzip wird so oft auf der aktuellen Spalte angewendet, bis es keine weiteren Möglichkeiten mehr gibt, Variablen zu streichen.

Im nächsten Schritt wird das gleiche Prinzip dann in der neuen Spalte angewendet, wobei nun 0, 1 und _ unterschieden werden (alle _ müssen hierbei übereinstimmen). Fortgeführt wird dies nun auch in den nachfolgend entstehenden Spalten, bis keine weitere Kombination mehr möglich ist. Die resultierenden Konjunktionen, die nicht abgedeckt/verwendet wurden, sind die Primterme der Ausgangsfunktion.

Danach werden die Variablen in ein Schema wie in Abschnitt 5.1 abgebildet eingetragen. Hier wird nun überprüft, welche Primterme wirklich gebraucht werden, also Kernimplikanten sind. Dazu wird zunächst getestet, ob ein Minterm von genau einem Primterm abgedeckt wird. Ist dies der Fall, muss dieser Primterm Kernimplikant sein.

Dann gibt es das Prinzip der Zeilendominanz. Dies bedeutet, dass wenn es zwei Primterme gibt, von denen einer alle vom anderen abgedeckten Minterme selbst abdeckt (oder sogar noch mehr), der zweite nicht benötigt wird, da er vom ersten dominiert wird.

Beispiel: Bei den beiden Primtermen $P_1 = m_1 + m_2$ und $P_2 = m_1$ (m_i sind die Minterme) wird P_2 von P_1 dominiert, muss also nicht weiter betrachtet werden.

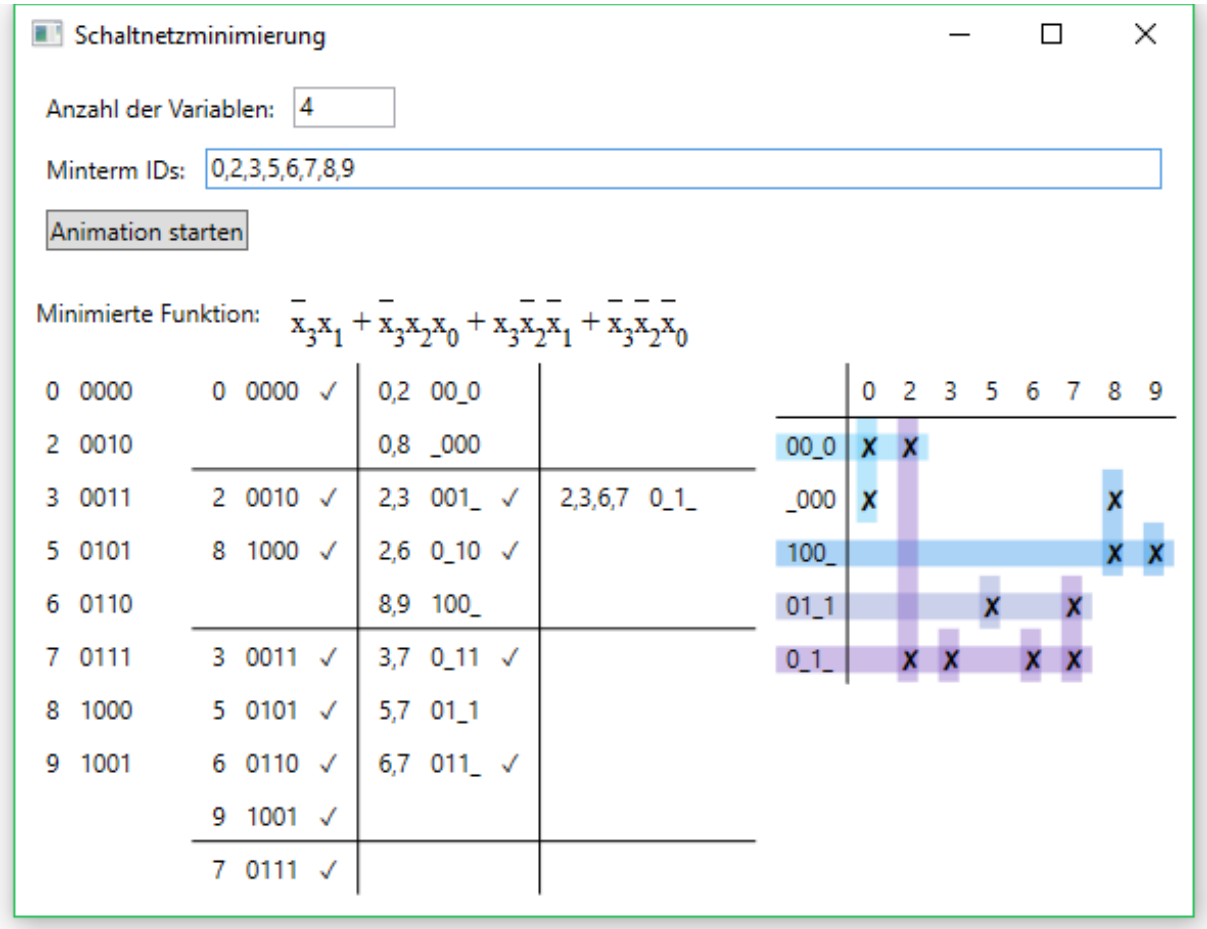
Zusätzlich gibt es die Spaltendominanz. Hier kann ein Minterm ignoriert werden, wenn es einen anderen gibt, dessen zugehörige Primterme eine Teilmenge der Primterme des ersten Minterms sind.

Beispiel: Bei den beiden Mintermen m_1 (von P_1 und P_2 abgedeckt) und m_2 (nur von P_1 abgedeckt) (P_i sind die Primterme) ist m_1 redundant, da er bei Abdeckung von m_2 auch abgedeckt werden muss.

Wenn alle Verfahren bereits benutzt wurden, kann dies wiederholt werden, allerdings auf einer Tabelle, in der redundante Prim- und Minterme bereits entfernt wurden. Die am Ende benötigten Primterme bilden eine minimierte Funktion.

5. Programm

5.1. Benutzung



Screenshot 1: Minimierung von Segment A eines BCD zu 7-Segment Decoders

Zunächst kann im ersten Eingabefeld die Anzahl der Variablen eingegeben werden. Es werden bis zu 30 Variablen unterstützt, allerdings ist das Verfahren nicht für diese Anzahlen geeignet, und entsprechend recht langsam (Das Verfahren benötigt im schlechtesten Fall eine exponentielle Laufzeit, aber bei bis zu 15 Variablen gab es keine Probleme mit der Geschwindigkeit). Im nächsten Feld werden die IDs der Minterme eingegeben, bei denen die Funktion wahr sein soll. Sobald etwas in einem der beiden Felder verändert wird, ändert sich auch die Ausgabe. Beide Tabellen werden entsprechend nach und nach mit einer Animation aktualisiert und die minimierte Funktion erscheint formatiert in der dritten Zeile.

5.2. Quellcode

Hinweis: Sämtliche Quelltexte sind privat unter GitHub[1] gespeichert. Zugriff darauf kann mithilfe der folgenden Anmeldedaten erlangt werden:

Benutzername: *Schaltnetzminimierung*
Passwort: *gdoCfC_JF17*

Das Programm wurde mit C# und WPF erstellt. Daher gibt es zum einen einzelne Klassen, die sich separat in eigenen Dateien befinden, und sogenannte Controls, also UI-Elemente, welche für das Layout noch eine zusätzliche .xaml-Datei besitzen.

Konjunktionen werden im Programm auf zwei Arten dargestellt: einfache Vollkonjunktionen (in denen also alle Variablen vorkommen) werden als vorzeichenlose 64-bit Zahl (ulong) gespeichert. Konjunktionen, in denen manche Variablen nicht mehr vorkommen, werden durch die Klasse *Konjunction* repräsentiert. In dieser gibt es ein Array, welches für jede Variable beinhaltet, ob diese einfach, negiert, oder gar nicht vorkommt.

Die kanonisch disjunktive Normalform einer Funktion wird als Instanz der Klasse *SOP* abgelegt. Diese beinhaltet die Anzahl der Variablen (*VariableCount*), eine Aufzählung aller Outputs abhängig von den Eingaben (*Outputs*), sowie ein Array von Konjunktionen, die die minimierte Funktion darstellen (*MinimizedFunction*).

Einstiegspunkt ist die Klasse *MainWindow*. Hier wird auf die Nutzereingaben reagiert und die Minimierung entsprechend gestartet und angezeigt. Die eigentliche Minimierung findet in der Control *QmcBooleanMinimizer* statt. Dieser wird zunächst eine kanonisch disjunktive Normalform übergeben, welche durch die Klasse *SOP* dargestellt wird.

```
39          public void AnimateMinimization()
40          {
41              output = new List<ListDictionary<byte, ListDictionary<Konjunction, bool>>>()
42                  { new ListDictionary<byte, ListDictionary<Konjunction, bool>>() };
43              minOnes = new List<byte>() { byte.MaxValue };
44              maxOnes = new List<byte>() { 0 };
```

Die eigentliche Minimierung findet nun in der Methode *QmcBooleanMinimiz*er.AnimateVisualization() statt. o*utput* stellt die linke Tabelle dar. Auf oberster Ebene gibt es eine Liste der einzelnen Spalten. Pro Spalte gibt es dann eine Aufzählung der einzelnen Zellen, die jeweils alle beinhalteten Minterme mit einer bestimmten Anzahl von Einsen beinhalten. In jeder Zelle gibt es dann eine Auflistung der dort vorkommenden Minterme, und für jeden Minterm wird

1 https://github.com/JonasWanke/CircuitMinimizer

gespeichert, ob dieser bereits genutzt wurde. *minOnes* und *maxOnes* beinhalten die minimale bzw. maximale Anzahl der Einsen in den Mintermen pro Spalte.

```
45              // Grid (left side)
46              List<ulong> cdnf = Sop.GetCdnf();
47              foreach (ulong mintermUl in cdnf)
48              {
49                  Konjunction minterm = new Konjunction(mintermUl, Sop.VariableCount);
50                  byte ones = GetOnesCount(minterm);
51                  if (!output[0].ContainsKey(ones))
52                      output[0][ones] = new ListDictionary<Konjunction, bool>();
53                  output[0][ones].Add(minterm, false);
54                  if (minOnes[0] > ones)
55                      minOnes[0] = ones;
56                  if (maxOnes[0] < ones)
57                      maxOnes[0] = ones;
58              }
59
60              InitGrid(output[0]);
```

Am Anfang wird in *cdnf* die KDNF (Auf Englisch *canonical disjunctive normal form*, daher die Abkürzung) der zu minimierenden Funktion abgefragt (Z. 46). In dieser kommen nur Minterme vor, daher sind diese als ulong gespeichert. Zunächst werden diese in Instanzen der Klasse *Konjunction* umgewandelt, damit alle in der Tabelle vorkommenden Minterme gleich behandelt werden können (Z. 49). Zeilen 51-52 sorgen nur dafür, dass eine neue Zelle angelegt wird, falls noch keine entsprechende existiert. Dieser wird dann in Zeile 53 der aktuelle Minterm hinzugefügt. minOnes und *maxOnes* werden in den Zeilen 54-57 aktualisiert, falls der aktuelle Minterm daran etwas ändern sollte. Ausgehend von diesen Startwerten wird nun die Visualisierung der linken Tabelle initialisiert (Z. 60).

```
62              // As long as konjunctions can be combined, go on column for column
63              bool changed = true;
64              byte currentIteration = 0;
65              while (changed)
66              {
67                  changed = false;
68                  minOnes.Add(byte.MaxValue);
69                  maxOnes.Add(0);
70
71                  // Search two successive rows
72                  for (byte i = minOnes[currentIteration]; i < maxOnes[currentIteration]; i++)
73                  {
74                      ListDictionary<Konjunction, bool> currentRow = output[currentIteration][i];
75                      ListDictionary<Konjunction, bool> nextRow = output[currentIteration][(byte)(i + 1)];
76                      if (currentRow == null || currentRow.Count == 0 || nextRow == null || nextRow.Count == 0)
77                          continue;
```

Nun wird Spalte für Spalte generiert, solange noch Minterme zusammengefasst werden können. *currentIteration* beinhaltet dabei die Nummer der Spalte, aus der die Konjunktionen aktuell kommen. Zunächst werden Standartwerte für die minimale und maximale Anzahl der Einsen in den Konjunktionen der nun erzeugten Spalte festgelegt (Z. 68-69). Danach wird über die einzelnen Zeilen der aktuellen Spalte

iteriert (Z. 72), und es werden jeweils zwei aufeinanderfolgende Zeilen betrachtet (Z. 74-75). Wenn davon eine leer sein sollte, wird dieser Fall übersprungen (Z. 76-77).

```
79                  // If there are two minterms that only differ in one place, combine them
80                  foreach (Konjunction first in currentRow.Keys)
81                      foreach (Konjunction second in nextRow.Keys)
82                          if (GetDifferentOnesCount(first, second) == 1)
83                          {
84                              Konjunction combination = Konjunction.GetCombination(first, second);
85                              byte ones = GetOnesCount(combination);
86                              if (minOnes[currentIteration + 1] > ones)
87                                  minOnes[currentIteration + 1] = ones;
88                              if (maxOnes[currentIteration + 1] < ones)
89                                  maxOnes[currentIteration + 1] = ones;
```

Anschließend werden alle möglichen Kombinationen von je einer Konjunktion aus der ersten und zweiten aktuellen Zeile betrachtet (Z. 80-81). Wenn diese sich in genau einer Stelle unterscheiden, können sie zusammengefasst werden, ansonsten wird die nächste Möglichkeit betrachtet (Z. 82). Als nächstes werden dann beide Konjunktionen zusammengefasst und in *combination* gespeichert (Z. 84). Von dieser Kombination wird nun die Anzahl der Einsen ermittelt, um zu bestimmen, in welcher Zeile diese neue Konjunktion abgelegt werden muss (Z. 85). Außerdem werden noch *minOnes* und *maxOnes* aktualisiert (Z. 86-89).

```
91                          // Create next column of minterms
92                          if (output.Count <= currentIteration + 1)
93                          {
94                              output.Add(new ListDictionary<byte, ListDictionary<Konjunction, bool>>());
95                              AddGridColumn();
96                          }
97                          // Create new cell
98                          if (output[currentIteration + 1][ones] == null)
99                          {
100                             output[currentIteration + 1][ones] = new ListDictionary<Konjunction, bool>();
101                             AddGridCell(currentIteration, ones);
102                         }
103                         //Add minterm itself
104                         if (!output[currentIteration + 1][ones].ContainsKey(combination))
105                         {
106                             output[currentIteration + 1][ones].Add(combination, false);
107                             AddMintermToGrid(currentIteration, ones, combination);
108                         }
109                         // Mark minterms as used
110                         UseMinterms(currentIteration, i, first, second);
111
112                         changed = true;
113                     }
114                 }
115                 currentIteration++;
116             }
```

In der ersten Abfrage wird nun überprüft, ob bereits die nächste Spalte in den Datenstrukturen vorhanden ist. Wenn nicht, wird diese erzeugt und angezeigt (Z. 94 bzw. 95). Danach wird ermittelt, ob die entsprechende Zelle schon existiert. Auch diese wird entsprechend angelegt und gezeigt (Z. 100 bzw 101). Als letztes wird dann noch die Konjunktion selbst hinzugefügt (Z. 106 bzw. 107). Die beiden verwendeten Konjunktionen werden dann in Zeile 110 als benutzt gekennzeichnet. *changed* wird auf wahr gesetzt, um zu zeigen, dass eine neue Konjunktion erzeugt

wurde, und es somit noch eine neue Spalte gibt. Als letztes wird die aktuelle Iteration um eins erhöht.

```
128             changed = true;
129             while (changed && primeImplicants.Count > 0 && uncoveredMinterms.Count > 0)
130             {
131                 changed = false;
```

```
123
124             List<Konjunction> primeImplicants = GetPrimeImplicants();
125             InitTable(cdnf, primeImplicants);
126             List<Konjunction> usedPrimeImplicants = new List<Konjunction>();
```

Nun wird die zweite Tabelle betrachtet. Es wird eine Liste der noch nicht abgedeckten Minterme angelegt (*uncoveredMinterms*, Z. 120-122). Außerdem werden aus der Ausgabe des vorherigen Teils alle Primterme gesucht, die noch nicht verwendet wurden, und in *primeImplicants* gespeichert (Z. 124). Der grafische Teil der Tabelle wird initialisiert (Z. 125) und es wird noch eine Liste der dann genutzten Primterme angelegt (Z. 126).

Nun werden die Primterme nacheinander ausgewählt, bis es keine Veränderungen mehr gibt (Z. 129).

```
133             // 1. Essential prime implicants
134             //    If a minterm is only covered by a single prime implicant it is essential.
135             for (int i = 0; i < uncoveredMinterms.Count; i++)
136             {
137                 ulong currentMinterm = uncoveredMinterms[i];
138
139                 List<Konjunction> associatedRows = GetAssociatedRows(currentMinterm, primeImplicants);
140                 if (associatedRows.Count == 1)
141                 {
142                     Konjunction primeImplicant = associatedRows[0];
143
144                     // Mark prime implicant used
145                     UsePrimeImplicant(primeImplicant, primeImplicants, usedPrimeImplicants);
146
147                     // Mark all other minterms covered by this prime implicant
148                     foreach (ulong minterm in primeImplicant.IncludedMinterms)
149                     {
150                         int index = uncoveredMinterms.IndexOf(minterm);
151                         if (index > -1 && index <= i)
152                             i--;
153                         uncoveredMinterms.Remove(minterm);
154                     }
155
156                     changed = true;
157                 }
158             }
```

Als erstes werden alle essentiellen Primterme ausgewählt. Dazu werden alle noch nicht abgedeckten Minterme betrachtet, und falls einer von nur einem Primterm abgedeckt wird, muss dieser verwendet werden (Z. 140). Dieser wird nun herausgesucht (Z. 142), benutzt (Z. 145), und alle weiteren Minterme, die von

diesem Primterm abgedeckt werden, werden markiert (Z. 148-154). Zuletzt wird noch *changed* wahr gesetzt, um zu zeigen, dass etwas sich geändert hat.

```
160             // 2. Row dominance
161             //    If the remaining minterms of a prime implicant are all included in another prime implicant, the current can be ignored.
162             for (int i = primeImplicants.Count - 1; i >= 0; i--)
163             {
164                 Konjunction first = primeImplicants[i];
165                 for (int j = 0; j < primeImplicants.Count; j++)
166                 {
167                     if (i == j)
168                         continue;
169
170                     Konjunction second = primeImplicants[j];
171                     IEnumerable<ulong> currentUncovered = uncoveredMinterms.Intersect(first.IncludedMinterms);
172                     if (currentUncovered.Any() && !currentUncovered.Except(second.IncludedMinterms).Any())
173                     {
174                         primeImplicants.Remove(first);
175
176                         changed = true;
177                     }
178                 }
179             }
```

Als nächstes wird das Verfahren der Zeilendominanz angewendet. Dazu werden alle möglichen Paare von Primtermen betrachtet (Z. 162-168), und falls einer alle Minterme eines anderen abdeckt (Z. 172), muss der abgedeckte nicht mehr betrachtet werden (Z. 174). Auch hier wird *changed* wieder aus dem gleichen Grund wahr gesetzt.

```
181             // 3. If nothing has changed, use the prime implicant which covers most of the remaining minterms
182             if (!changed)
183             {
184                 Konjunction primeImplicant = primeImplicants.OrderByDescending((minterm) => uncoveredMinterms.Intersect(minterm.IncludedMinterms).Count()).First();
185                 UsePrimeImplicant(primeImplicant, primeImplicants, usedPrimeImplicants);
186
187                 // Mark all other minterms covered by this prime implicant
188                 foreach (ulong minterm in primeImplicant.IncludedMinterms)
189                     uncoveredMinterms.Remove(minterm);
190
191                 changed = true;
192             }
193         }
```

Falls bisher nichts passiert ist, wird der Primterm genommen, der am meisten Minterme abdeckt (Z. 184-185). Hier werden auch alle weiteren Minterme markiert, die durch diesen abgedeckt werden. Zuletzt wird *changed* wieder wahr gesetzt.

```
195             Sop.MinimizedFunction = usedPrimeImplicants.ToArray();
196         }
```

Als letztes wird dann die minimierte Funktion bei der ursprünglichen KDNF gespeichert (Z. 195).

6. Ausblick

Später können auch sogenannte Don‘t Cares genutzt werden, also Eingänge, bei denen es egal ist, was rauskommt (Beispiel: Ein BCD zu 7-Segment Decoder muss bei einem zweistelligen Eingang nichts sinnvolles anzeigen). Damit einhergehend wird die Eingabe noch überarbeitet, sodass man in einer Wahrheitstabelle angeben kann, was passieren soll.

Das Verfahren der Spaltendominanz wird noch implementiert.

Auch sollen die Animationen noch überarbeitet werden.