Optimierte Zustandscodierung

Algorithmen und Datenstrukturen

David Metzger Manuel Preuschoff Patrick Wenninger

8. Juni 2018

Inhaltsverzeichnis

1 Aufgabenstellung **3**
1.1 Anforderungen . . . 3
1.1.1 Eingangsdaten . . . 3
1.1.2 Verarbeitung . . . 3
1.1.3 Ausgangsdaten . . . 3
1.1.4 Vergleich . . . 4

2 Unterteilung des Programms in Blöcke **4**
2.1 Einlesen der Eingabedaten . . . 4
2.2 Datenstruktur für Automatentabelle . . . 5
2.3 Zustanscodierungsoptimierer . . . 6
2.4 Ansteuertabelle für Minilog . . . 7
2.5 Zählen der Gatter . . . 7

3 Struktogramm **8**
3.1 Optimierer: Hauptprogramm . . . 8
3.2 Optimierer: Parser . . . 9
3.3 Optimierer: hohe Priorität . . . 10
3.4 Optimierer: mittlere Priorität . . . 11
3.5 Optimierer: niedrige Priorität . . . 12
3.6 Optimierer: Untermengen entfernen . . . 12
3.7 Optimierer: Codierung optimieren . . . 13

4 Laufzeitanalyse **14**
4.1 Laufzeitkomplexität . . . 14
4.1.1 highPriority() . . . 14
4.1.2 meanPriority() . . . 14
4.1.3 lowPriority() . . . 14
4.1.4 optimize() . . . 14
4.2 Zusammenfassung . . . 14
4.3 Messungen . . . 14

5 Test und Beispiele **15**
5.1 Kleines Beispiel . . . 15
5.2 Mittleres Beispiel . . . 16
5.3 Großes Beispiel . . . 18

1 Aufgabenstellung

Die Codierung der Zustände in Zustandsmaschinen kann bei der Realisierung der Ansteuernetzwerke je nach Wahl der Codes zu unterschiedlich großen Gatterzahlen führen. Es sollen benachbarte Zustände und die Übergänge zwischen ihnen betrachtet werden.

Mit Heuristiken lässt sich eine günstige Anzahl von Gatterzahlen erreichen. Es soll das Verfahren der minimalen Übergangsdistanz angewendet werden um eine Codierung zu finden, bei der benachbarte Zustände auch benachbarte Codes besitzen.

1.1 Anforderungen

1.1.1 Eingangsdaten

Eingabedaten sollen aus einer Datei gelesen werden. Die Eingabedaten beschreiben die Übergänge und sind in der folgenden Form angeben:

```
1 Begin
2 DEFSTATE a,b,c; DEFIN i,j,k; DEFOUT c,d;
3 /*[Inputs]=(Inp.Values)(act.state)>[Outputs]:(Outp.Values)(next.state)*/
4
5 [i,j]=(0,1)(a)     > [c]:(0)(a)     /* k ist hier implizit mehrdeutig */
6 [i,j,k]=(0,x,1)(a) > [c,d]:(0,1)(b) /* j ist hier explizit mehrdeutig */
7 End
```

1.1.2 Verarbeitung

Die Zustandscodierung der eingelesenen Zustandsmaschine wird nach der minimalen Übergangsdistanz optimiert. Die Optimierung wird in Abschnitt 2.3 beschrieben.

1.1.3 Ausgangsdaten

Ausgabedaten sollen zwei Dateien für das Logikminimierungsprogramm Minilog sein. (binäre und optimierte Zustandscodierung) Aus dem Zustandsdiagramm lässt sich mit einer Codierung der Zustände eine Ansteuertabelle erzeugen. Die Ansteuertabelle soll dem Eingabeformat von Minilog entsprechen und sieht folgendermaßen aus:

```
1  table Tabellenname
2  input A B C
3  output Y Z
4  " Kommentar
5  000 | 00
6  001 | 10
7  010 | 10
8  011 | 01
9  100 | 10
10 101 | 01
11 110 | 01
12 111 | 11
13 end
```

1.1.4 Vergleich

Ein Vergleich zwischen einer zufällig vorgenommenen Codierung und mit einer Codierung über die heuristischen Vorschriften erfolgt mit einem eigens entwickelten Gatterzähler, welcher die Minimierten Tabellen von Minilog einliest, die Schaltnetze analysiert und ausgibt und die benötigten Gatter (zwei Eingänge, ein Ausgang ; AND und OR) zählt.

2 Unterteilung des Programms in Blöcke

2.1 Einlesen der Eingabedaten

Die Lexikalische Analyse besteht aus einem Scanner und einem Parser. Der Scanner zerlegt den Eingabestrom in Token und übergibt diese Information an den Parser. Dieser Analysator kann vordefinierte Schlüsswelwörter erkennen und auch spezifieren, welche Art von Token vorliegt (Identifier, String oder Integer).

Die Verarbeitung der Token geschieht im Parser. Dieser wird vom Main-Programm initialisiert und bekommt die Eingabedatei übergeben. Beim Aufruf des Parsers wird das Tabellenobjekt übergeben, in dem er die eingelesenen Daten abspeichert. Das Verarbeiten der Eingabedaten wird mit einer Zustandsautomaten durchgeführt. Der aktuelle Zustand wird in der lokalen Variable `state` vom Datentyp Enumeration (Aufzählung) gespeichert.

Folgende Zustände sind in der Enumeration `parstates` definiert:

P_HEADER	Alles vor "`Begin`"wird ignoriert
P_DEFSELECT	Auswahl der nächsten Definition
P_DEFSTATE	Einlesen der Definition der Zustände
P_DEFIN	Einlesen der Definition der Eingänge
P_DEFOUT	Einlesen der Definition der Ausgänge
P_READLINEINPUTS	Einlesen der Eingänge der Übergangsdefinition
P_READLINEINVALS	Einlesen der Eingangswerte der Übergangsdefinition
P_READLINESSTATE	Einlesen des Ausgangszustands der Übergangsdefinition
P_READLINEOUTPUTS	Einlesen der Ausgänge der Übergangsdefinition
P_READLINEOUTVALS	Einlesen der Ausgangswerte der Übergangsdefinition
P_READLINEDSTATE	Einlesen des Zielzustandes der Übergangsdefinition
P_ERROR	Fehlerstatus, meldet Fehler mit Zeile und bleibt hier

In den Zuständen werden die jeweiligen Funktionen zur Verarbeitung der Token aufgerufen. Zu Beginn wir der erste Zustand P_HEADER gesetzt, die folgenden Zustände werden von den Zustandsfunktionen gesetzt. Der Zustandsautomat fragt mit einer switch-case ab, welcher Zustand vorliegt und führt die entsprechenden Funktionen aus.

Im Zustand P_HEADER wird der Header übersprungen, wobei der Header maximal 100 000 Wörter lang sein darf.

P_DEFSELECT entscheidet anhand vom Schlüsselwort, ob als nächstes die Definition der Zustände, der Eingänge oder der Ausgänge eingelesen werden und setzt dementsprechend die Variable `state` (P_DEFSTATE, P_DEFIN und P_DEFOUT). Wurden alle Definitionen durchlaufen, wird in der Eingangsdatei die Definition der Zustandsübergänge erwartet.

Diese wird zeilenweise von den Zuständen P_READLINEXXX eingelesen und verarbeitet. In diesen Zuständen werden die Zustandsübergänge nach und nach in Variablen gespeichert. Wird nach dem letzten P_READLINEXXX-Zustand (P_READLINEDSTATE)

wieder der erste P_READLINEXXX-Zustand (P_READLINEINPUTS) gesetzt, sind alle Variablen aktuell und die `link`-Funktion wird aufgerufen. Sie trägt die gesammelten Informationen in die Automatentabelle ein. Danach müssen die Variablen für das Einlesen der nächste Zeile gelöscht werden.

Zuletzt gibt es noch den Fehlerzustand P_ERROR, der immer dann gesetzt wird, wenn die Eingabedatei fatale Fehler enthält. Dieser verhindert, dass das Programm die Eingangsdatei fehlinterpretiert. Zu fatalen Fehlern zählen zum Beispiel aufzählungszeichen beim Folgezustand, weil an dieser Stelle nur ein Identifier erwartet wird.

2.2 Datenstruktur für Automatentabelle

Zur Darstellung der Zustandsmaschinentabelle wurde die Klasse `smtable` entworfen. Um die Zustandstabelle für den Benutzer verständlich zu speichern, verwenden wir die Datenstruktur `map< string, vector< entry > >`. `entry` ist eine C-Struktur,bestehend aus den Strings `next_state` und `out_list`. Es stellt genau ein Feld in der Zustandstabelle dar. Mittels `typedef` haben wir dieser Datenstruktur das Schlüsselwort `tabletype` zugewiesen. In der Klasse `smtable` liegt eine Instanz mit dem Namen 'table' von diesem Tabellentyp vor. Darin liegt die gesamte Information der Zustandsübergänge mit Ausgangssignalen.

Des Weiteren beinhaltet `smtable` Ganzzahlen für die Breite und Höhe der Tabelle. Eingangssignale, Ausgangssignale und Zustände sind in Variablen des Typs `elementlist` abgelegt. `elementlist` ist der umdefinierte Typ `vector<string>`.

Um die Tabelle zu bearbeiten sind in der Klasse verschiedene Funktionen definiert. Mit den setXX Funktionen (setStates, setInputs und setOutputs) werden alle genannten Zustände, Eingänge und Ausgänge abgespeichert. Diese können aber auch mit der Initialisierung übergeben werden.

Die Initialisierungsfunktion bereitet die Tabelle vor: Es werden vorbesetzte Zeilen (Dummy-Werte) mit den Indizes (Zustände) entsprechend der Zustandsliste angehängt. Diese vorbesetzte Zeilen (zweite Argumene der `map<>`-Einträge), also die `vector< entry >` entsprechen den einzelnen Zeilen der Tabelle. Der `vector<>` wird in der entsprechenden Breite angelegt. Diese Breite entspricht der Anzahl der Spalten der Tabelle, also die Anzahl der Eingangskombinationen. Jeder `next_state`-String in `entry` bleibt leer und die Ausgangswerte werden zu "don't cares"('x') gesetzt.

Die `link`-Funktion ist das Herzstück der Klasse. Sie trägt einen Zustandsübergang in die Zustandstabelle ein. Die Besonderheit daran ist, dass Mehrdeutigkeiten verarbeitet werden. Sie besitzt sechs Eingangsargumente: Die Eingangsnamensliste, die zugehörigen Eingangswerte, den Ausgangszustand, die Ausgangsnamensliste, die zugehörigen Ausgangswerte und den Zielzustand. Die Werte werden nach der Reihenfolge der Namensliste zugeordnet. Diese Liste muss weder in der richtigen Reihenfolge, noch vollständig sein. Nicht genannte Namen werden als Mehrdeutigkeit interpretiert ('x' an entsprechender Stelle). Ebenso können die Mehrdeutigkeiten konkret bei der Wertevariable als 'x' angegeben werden. Wird beispielsweise für eine Tabelle mit zwei Eingängen nur ein Eingang bei der `link`-Funktion übergeben, muss der Zielzustand mit Ausgangswerten an zwei Felder in der Zustandstabelle eingetragen werden.

Zusätzlich zu den benötigten Zugriffsfunktionen ist eine `print`-Funktion definiert. Diese gibt die Zustandstabelle an der Konsole in bekannter Form aus.

In den verschiedenen Zugriffsfunktionen werden zur Lesbarkeit des Quellcodes Unterfunktionen benötigt:

`int2bit( val, width )` wandelt einen Integer-Wert in einen binären String. Es wird der Integer-Wert und die Länge des gewünschten Strings übergeben. Rückgabewert ist ein Zeiger auf den generierten string. Dieser muss nach Verarbeitung manuell gelöscht werden.

`bitsMatch( a, b )` vergleicht eine Ganzzahl **a** und einen String **b** auf bit-Ebene. Der String muss aus den ASCII-Zeichen '0', '1' und 'x' bestehen. Die Länge ist beliebig. Sie vergleicht die einzelnen Bits des Integer-Wert mit dem binären String von rechts beginnend. Können alle '1' und '0' in **b** mit entsprechenden Bits in **a** abgeglichen werden, gibt die Funktion den boolschen Wert zurück, sonst `false`. Beispiel: `bitsMatch( 3 , "01x" )` gibt `true` zurück.

2.3 Zustanscodierungsoptimierer

Die Zustandscodierungsoptimierung soll mit dem Verfahren der minimalen Übergangsdistanz durchgeführt werden. Bei der Anwendung des Verfahrens wird versucht eine Codierung zu finden, bei der benachbarte Zustände auch benachbarte Codes besitzen. Dazu müssen drei Fälle mit unterschiedlichen Prioritäten abgeprüft werden:

-hohe Priorität
-mittlere Priorität
-niedere Priorität

Hohe Priorität liegt vor, wenn für eine gegebene (beliebige) Eingangskombination zwei Zustände einen gemeinsamen Nachfolgeknoten haben. Alle Zustände, auf die die hohe Priorität zutrifft werden in einer `map< string , vector< string > >` abgelegt.

Mittlere Priorität liegt vor, wenn zwei Zustände von einem gemeinsamen Knoten erreicht werden können. Die Eingangskombination wird bei mittlerer Priorität nicht betrachtet. Auch die mittleren Prioritäten werden in einer `map< string , vector< string > >` abgelegt.

Niedere Priorität liegt bei Zuständen vor, die ein gemeinsames Ausgangsverhalten aufweisen. Das heißt bei gleicher Eingangskombination geben die Zustände auch die gleiche Ausgangskombination aus. Im Code haben wir alle niederen Prioritäten in einem `vector< vector< string > >` abgelegt. Bei der niederen Priorität können auch zwei gleiche Prioritäten vorkommen. Doppelte Prioritäten und Untermengen werden mit der Funktion `removeSubsets()` gelöscht. Grundsätzlich werden Zustandskombinationen, die andere Zustandskombinationen enthalten bevorzugt und der Rest gelöscht. Beispiel: Tritt die Kombination A B C und A B auf, so wird A B gelöscht.

Hohe Priorität hat den größten Einfluss für das Einsparen von Gattern und muss daher als erstes angewendet werden. Danach wird die mittlere Priorität angewandt und erst danach die niedere Priorität. Die Reihenfolge der Zustandscodierung speichern wir in einem `vector< string >` ab. Als Zustandscodierung nehmen wir den Gray-Code. Die Variable set_states gibt an wie viele Zustände schon gesetzt sind. Vor jedem Anwenden der Prioritäten wird überprüft, ob schon alle Zustände gesetzt sind.

Ebenso wird beim Setzen überprüft, ob es eine hohe Priorität von den Zuständen von einer hohen Priorität gibt. Dann werden diese Zustände gegebenenfalls vertauscht angeordnet. Bei der Anwendung von niedrigeren Prioritäten werden jeweils nur Zustände beachtet, die nicht schon bei höheren Prioritäten verwendet wurden. Es wird geschaut, ob es mindestens 2 Zustände in niederen Prioritäten gibt, die noch nicht gesetzt sind. Wenn es welche gibt, werden diese untereinander in dem Zustandscodierungs-Vektor ergänzt. Am Ende werden noch alle übrigen Zustände gesetzt.

2.4 Ansteuertabelle für Minilog

Das Hauptprogramm des Optimierers muss die optimierte- und zum Vergleich auch die nichtoptimierte Zustandscodierung nun in zwei Dateien für Minilog ausgeben. Dafür wird die überladene Funktion `writeOutputFile()` verwendet. Wird ihr nur ein Objekt der Klasse `smtable` übergeben, schreibt sie eine Datei namens `ZMnichtoptimiert.tbl` in das Arbeitsverzeichnis. Übergibt man vor dem Objekt eine `elementlist` (Typ: `vector< string >`) erzeugt `writeOutputFile()` die Datei `ZMoptimiert.tbl`.

Die Dateien werden nach der Vorgabe von Minilog erstellt und enthalten die Zustandsübergangstabelle. Bits der aktuellen Zustandscodierung heißen `CXX` und dazugehörige Bits der Folgezustände heißen `DXX`. Es wurden zwei Dezimalstellen zur Benamung gewählt, da man mit maximal 100 Bits $2^{100} \approx 10^{30}$ Zustände codiert werden können.

2.5 Zählen der Gatter

Aus der Ausgabedatei von Minilog kann die benötigte Gatteranzahl und die Logikgleichungen ermittelt werden. Die Ausgabedatei von Minilog ist immer gleich aufgebaut. Nach der vierten Kette von Gleichzeichen (`==`) kommen die Eingangs- und Ausgangsvariablen. Die Namen der Eingangssignale und Ausgangssignale werden in einem Vektor abgelegt. Nach der fünften Kette von Gleichzeichen (`==`) kommt die optimierte Ansteuertabelle. Dann wird die Ansteuertabelle Zeile für Zeile durchlaufen. In jeder Zeile zählen wir bei den Eingangsvariablen alle '`0`' und '`1`'. Diese Anzahl minus eins addieren wir zu der Gesamtgatteranzahl hinzu. Anzahl minus eins kommt daher, da bei zwei Variablen nur ein Gatter erforderlich ist. In der ersten Zeile legen wir für die Größe der Ausgangsvariablen einen Vektor an. In diesem Vektor speichern wir die Anzahl der '`1`' bei den Ausgangsvariablen (spaltenweise) ab. Zu der Gesamtgatteranzahl kommen am Ende noch die Anzahl der Ausgangsvariablen minus eins hinzu. Beim Durchlaufen von jeder Zeile werden die Logikgleichungen erzeugt und in einem String abgelegt. Wenn eine '`1`' bei den Ausgangsvariablen ist, wird der String mit der Logikgleichung in dem `vector< string >` bei der Ausgangsvariablen angehängt.

3 Struktogramm

3.1 Optimierer: Hauptprogramm

Das Hauptprogramm erwartet beim Aufrufen mit Option an der ersten Stelle den vollen Dateinamen mit Endung. Wird die Ausführbare ohne Option aufgerufen, muss der Benutzer an der Konsole den Dateinamen eingeben. Die Dateiendung "`.txt`"wird angehängt.

Nach erfolgreichem Öffnen der Datei wird der Parser Initialisiert und aufgerufen. Enthält die Tabelle fatale Fehler, wird dieser mit Zeilenangabe innerhalb des Parsers gemeldet und das Einlesen gestoppt. Dieser Fehler wird dem Hauptprogramm zurückgegeben.

Nun wird die Tabelle ausgegeben. Im Fehlerfall ist diese unvollständig.

Die Optimierung wird auch nur bei fehlerfreiem Einlesen durchgrführt. Nacheinander werden die Prioritätsfunktionen aufgerufen und die rückgegebenen Listen gespeichert. Nach dem Aufräumen der niedrigen Priorität, erhält man die optimierte Reihenfolge der Zustände von der Optimierfunktion. Nun werden die Dateien für Minilog geschrieben.

Wurde die Ausführbare ohne Option gestartet oder lag ein Fehler vor wird die Konsole offen gehalten, sodass der Benutzer die Ausgabe betrachten kann. Zuletzt wird der Fehlerstatus an Windows weitergegeben.

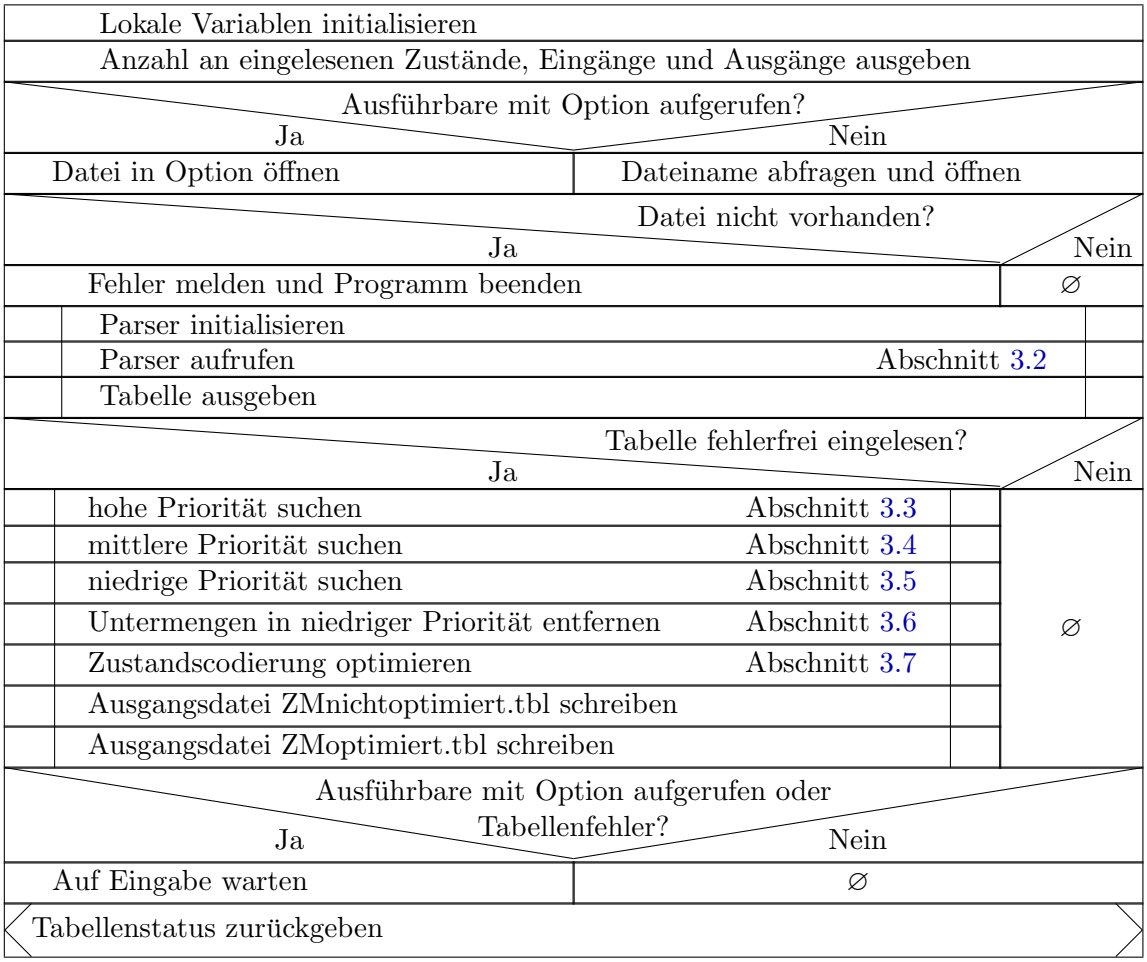

Abbildung 1: Programmablauf Hauptprogramm: main(argc, argv)

3.2 Optimierer: Parser

Wie bereits erwähnt besteht der Parser aus einer Zustandsmaschine, welche bei jedem Durchlauf das nächste Token des Datenstroms über `yylex()` bekommt.

Abbildung 2: Programmablauf Parser: CParser::yyparse(smtable &table)

Alle Daten vor `Begin` werden im Zustand `P_HEADER` ignoriert. Danach entscheidet `P_DEFSELECT` welche Definition folgt und verweist auf die entsprechenden Zustände `P_DEFSTATE`, `P_DEFIN` oder `P_DEFOUT`. Diese geben beim Ende der Definition wieder an `P_DEFSELECT` zurück. Sind alle Definitionen erledigt, erwartet der Parser eine Übergangsdefinition. Diese wird mit den `P_READLINEXXX`-Zustände nacheinander in lokale Variablen eingelesen. Ist das Einlesen einer Zeile beendet (wenn Verweis an `P_READLINEINPUTS`), werden die zwischengespeicherten Daten in die Tabelle eingetragen und die Variablen zurückgesetzt.

Mit dem Schlüsselwort `End` wird die Zustandsmaschine verlassen. An der Konsole erscheinen die Anzahlen der erkannten Zustände, Eingänge und Ausgänge. Zuletzt wird der Fehlerstatus zurückgegeben.

3.3 Optimierer: hohe Priorität

Die hohe Priorität geht Spalte für Spalte von der Zustandstabelle durch. Alle Folgezustände einer Spalte (gleiche Eingangskomination) werden mit jedem anderen Zustand (jeder anderen Zeile) verglichen. Ist der Vergleich positiv, wird dieser Kandidat an eine temporäre Liste angehängt. Enthält diese Liste nach allen Vergleichen mehrere Kandidaten, wird sie als hohe Priorität gespeichert. Danach wird die temporäre Liste gelöscht und bei der nächsten Zeile wieder begonnen.

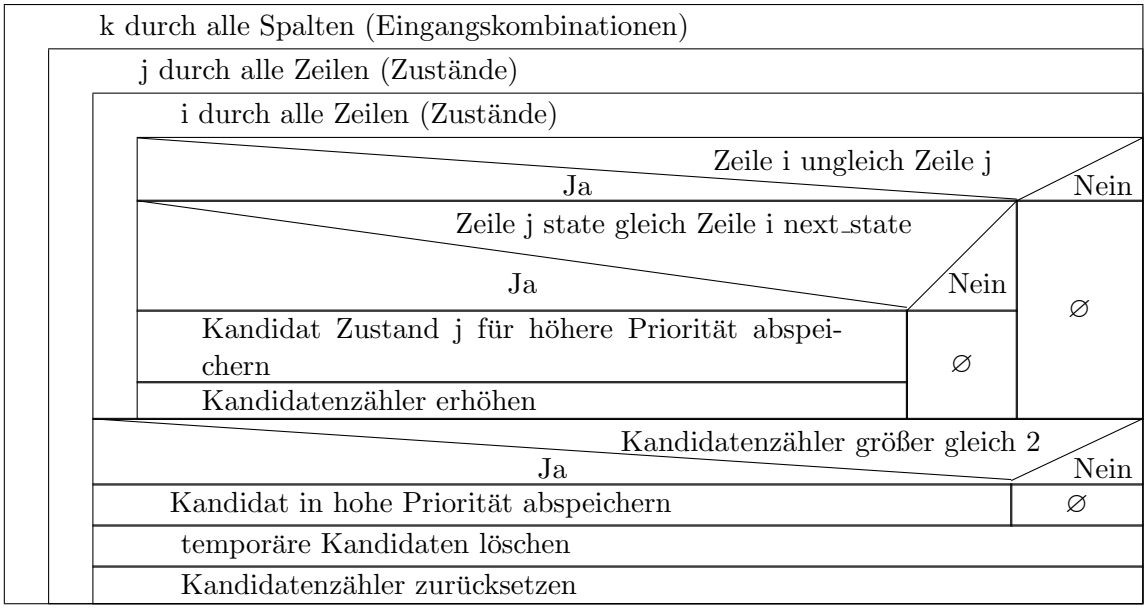

Abbildung 3: Programmablauf Optimierer: highPriority(smtable &table)

3.4 Optimierer: mittlere Priorität

Die mittlere Priorität geht Zeile für Zeile von der Zustandstabelle durch. Alle Folgezustände einer Zeile werden mit dem Zustand in dieser Zeile verglichen. Ist der Vergleich positiv, wird dieser Kandidat an eine temporäre Liste angehängt. Enthält diese Liste nach allen Vergleichen mehrere Kandidaten, wird sie als mittlere Priorität gespeichert. Danach wird die temporäre Liste gelöscht und bei der nächsten Zeile wieder begonnen.

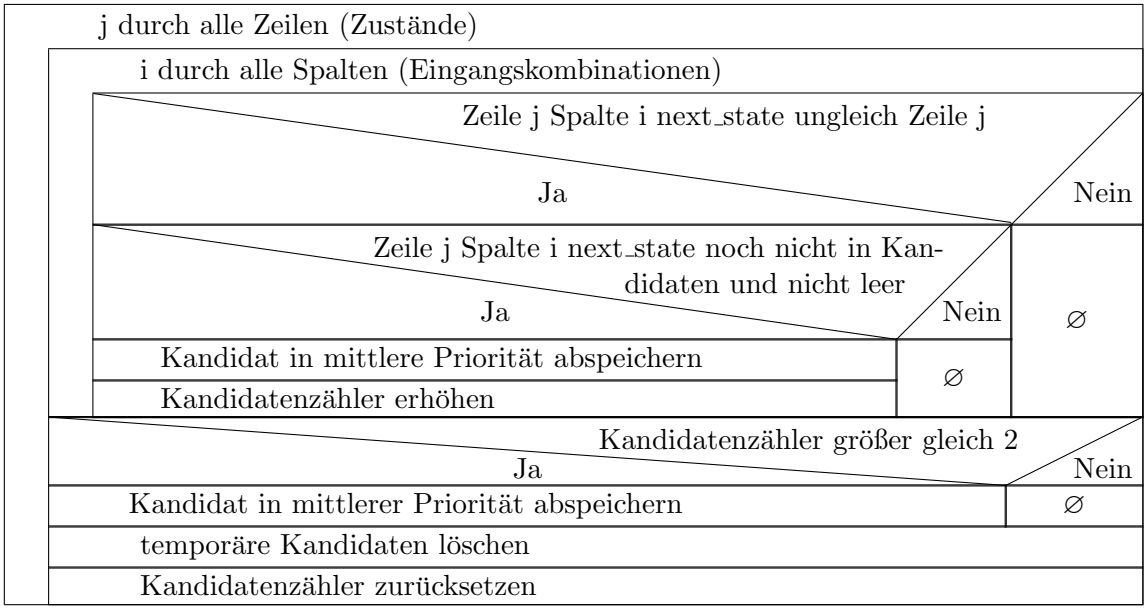

Abbildung 4: Programmablauf Optimierer: meanPriority(smtable &table)

3.5 Optimierer: niedrige Priorität

Die niedrige Priorität geht Spalte für Spalte von der Zustandstabelle durch. Alle möglichen Ausgangskombinationen werden mit den Ausgangswerten verglichen. Ist der Vergleich positiv, wird dieser Kandidat an eine temporäre Liste angehängt. Enthält diese Liste nach allen Vergleichen mehrere Kandidaten, wird sie als niedrige Priorität gespeichert. Danach wird die temporäre Liste gelöscht und bei der nächsten Spalte wieder begonnen.

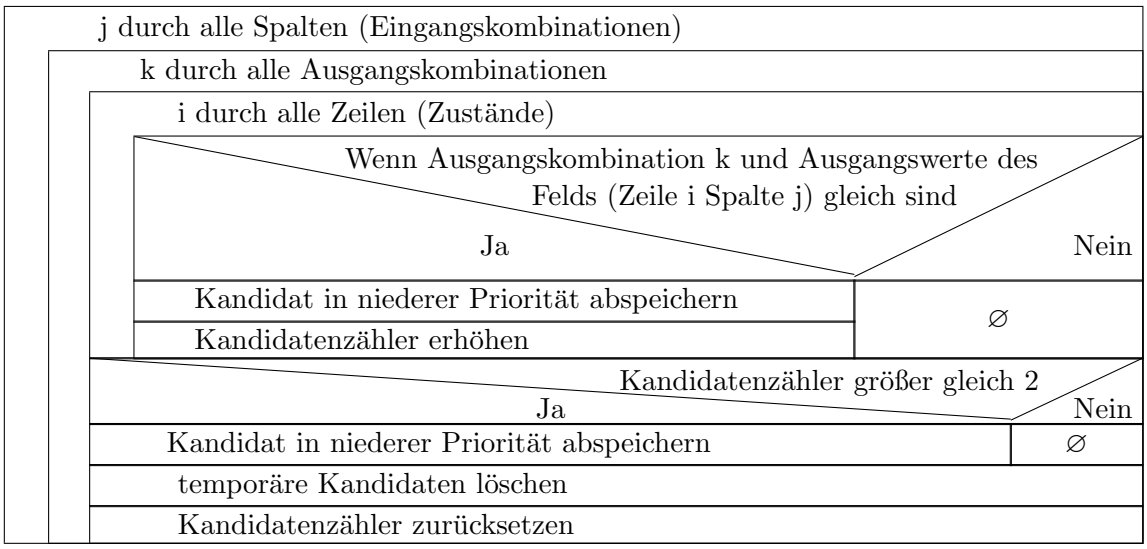

Abbildung 5: Programmablauf Optimierer: lowPriority(smtable &table)

3.6 Optimierer: Untermengen entfernen

Die Funktion `lowPriority(table)` erstellt einen `vector< vector< string > >`. Diese Liste enthält Kombinationen aus Zuständen, welche niedrige Priorität erfüllen. Diese Kombinationen können mehrfach auftreten und auch Teilmengen voneinander sein. Diese Funktion löscht überflüssige Kombinationen, indem Zeile für Zeile mit jeder anderen Zeile verglichen wird.

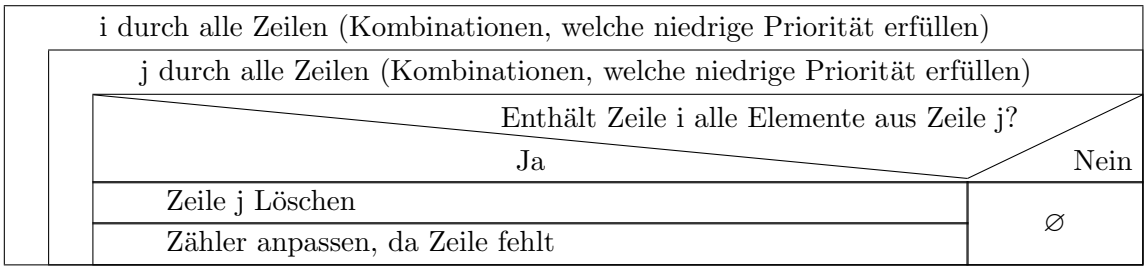

Abbildung 6: Programmablauf Parser: removeSubsets(lowpriotype low_priority)

3.7 Optimierer: Codierung optimieren

Der Optimierer gibt eine optimierte Liste an Zustandsnamen zurück. Es werden alle gefundenen Prioritäten übergeben und priorisierte Zustände nebeneinander gesetzt. Werden diese Zustände im Gray-Code inkrementell codiert, erhält man eine optimierte Codierung.

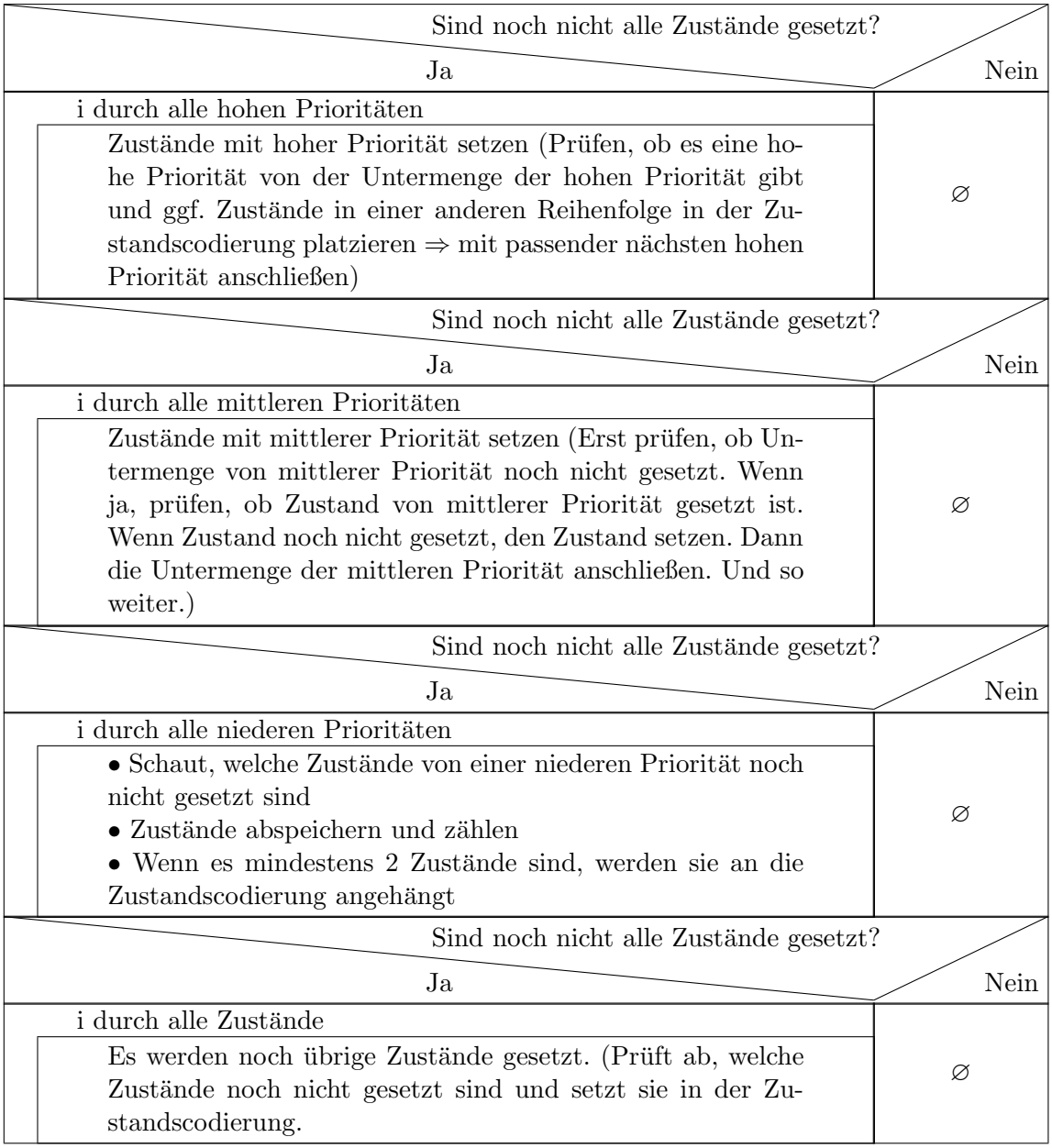

Abbildung 7: Programmablauf Optimierer: optimize(high_priority, mean_priority, low_priority, table)

4 Laufzeitanalyse

4.1 Laufzeitkomplexität

Ins Gewicht fallen M (Anzahl Zustände), N (Anzahl Eingänge) und O (Anzahl Ausgänge).

4.1.1 highPriority()

Die hohe Priorität läuft durch alle Eingangskombinationen. Die Anzahl dieser steigt im Quadrat mit der Anzahl der Eingänge. Innerhalb der ersten Schleife werden alle Zustände doppelt durchlaufen.

Die Laufzeitkomplexität entspricht $M^2 \cdot N^2$

4.1.2 meanPriority()

Mittlere Priorität durchläuft jede Zelle der Übergangstabelle ein mal.

Die Laufzeitkomplexität entspricht $M \cdot N^2$

4.1.3 lowPriority()

Niedrige Priorität durchläuft alle Eingangskombinationen, darin alle Ausgangskombinationen und innerhalb dieser Schleife alle Zustände.

Die Laufzeitkomplexität entspricht $M \cdot N^2 \cdot O^2$

4.1.4 optimize()

Die größte Schleife der Optimierung durchläuft die gefundenen Prioritätenlisten, weswegen keine Laufzeitkomplexität in Abhängigkeit von M, N und O gegeben werden kann.

4.2 Zusammenfassung

Auf die gesamte Laufzeitkomplexität wirken sich jeweils die höchsten Potenzen der Anzahlen aus.

Die Laufzeitkomplexität entspricht mindestens $M^2 \cdot N^2 \cdot O^2$

4.3 Messungen

Tabelle 1: Laufzeitvergleich

Tabellennamen	Anzahl Zustände	Anzahl Eingangssignale	Anzahl Ausgangssignale	Laufzeit
kleine Automatentabelle	3	2	1	2ms
mittlere Automatentabelle	4	2	1	4ms
riesige Automatentabelle	15	1	1	8ms

Die Laufzeit der verschiedenen Automatentabellen ist abhängig von der Anzahl der Zustände, Eingangssignale und Ausgangssignale. Außerdem ist es auch stark davon abhängig welche Prioritäten bei den Zuständen vorliegen.

5 Test und Beispiele

5.1 Kleines Beispiel

Als Beispiel für die Anwendung des Algorithmus auf eine kleine Automatentabelle wurde ein Automat mit drei Zuständen und aussagekräftige Namen gewählt.

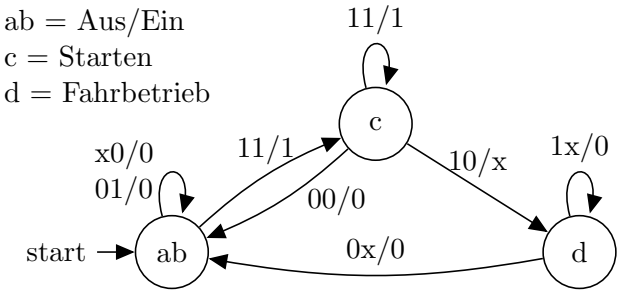

Abbildung 8: Zustandsdiagramm kleines Beispiel

Das kleine Beispiel beschreibt eine Zustandsmaschine eines Anlassers. Die Eingangssignale sind Zündung und Anlassen. Ausgang ist der Anlasser. Die Zustandsmaschine verhindert mit einem dritten Zustand das erneute starten des Anlassers.

```
Anzahl Zustaende: 3
Anzahl Eingangssignale: 2
Anzahl Ausgangssignale: 1
Reihenfolge Eingangssignale: Zuendng Anlassn
Reihenfolge Ausgangssignale: Anlassr

|       |00     |01     |10     |11     |
|-------|-------|-------|-------|-------|
|Aus_Ein|Aus_Ein|Aus_Ein|Aus_Ein|Starten|
|Starten|Aus_Ein|Aus_Ein|Fahrbtr|Starten|
|Fahrbtr|Aus_Ein|Aus_Ein|Fahrbtr|Fahrbtr|
|-------|-------|-------|-------|-------|
|Aus_Ein|0      |0      |0      |1      |
|Starten|0      |0      |x      |1      |
|Fahrbtr|0      |0      |0      |0      |

Ausgabedatei "ZMnichtoptimiert.tbl" erfolgreich geschrieben
Ausgabedatei "ZMoptimiert.tbl" erfolgreich geschrieben
```

Nach Optimierung der Zustandscodierung nach dem Verfahren der minimalen Übergangsdistanz und Betrachtung der drei Fälle ergibt sich folgende Zuordnung von Zuständen zu Codes. Als Code wurde der Gray-Code gewählt.

Tabelle 2: Optimierte Codierung

Zustände	Codierung
Aus/Ein	00
Starten	01
Fahrbetrieb	11

Der Vergleich zwischen binärer Codierung und optimierter Codierung im Gray-Code zeigt für diesen Automaten mit drei Zuständen keine Verbesserung. Die Optimierung zeigt erst für größere Automaten ihre Auswirkungen.

```
Gatteranzahl von '../../zmnichtoptimiert.min': 6
Gatter:
D00 = (ZUENDNG ^ C00) v (ZUENDNG ^ ANLASSN' ^ C01)
D01 = (ZUENDNG ^ ANLASSN ^ C00')
Gatteranzahl von '../../zmoptimiert.min': 6
Gatter:
D00 = (ZUENDNG ^ C00) v (ZUENDNG ^ ANLASSN' ^ C01)
D01 = (ZUENDNG ^ ANLASSN) v (ZUENDNG ^ ANLASSN' ^ C01)
```

5.2 Mittleres Beispiel

Das mittlere Beispiel beschreibt eine Zustandsmaschine, die ein taktvorderflankengesteuertes D-Flip-Flop beschreibt. Eingangssignale sind das Taktsignal und den D-Eingang. Ausgang ist der nichtinvertierte D-Ausgang.

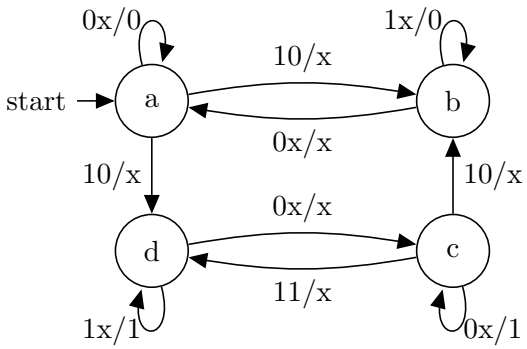

Abbildung 9: Zustandsdiagramm mittleres Beispiel

```
Anzahl Zustaende: 4
Anzahl Eingangssignale: 2
Anzahl Ausgangssignale: 1
Reihenfolge Eingangssignale: D C
Reihenfolge Ausgangssignale: Q

|       |00     |01     |10     |11     |
|-------|-------|-------|-------|-------|
|a      |a      |b      |a      |c      |
|b      |a      |b      |a      |b      |
|c      |d      |c      |d      |c      |
|d      |d      |b      |d      |a      |
|-------|-------|-------|-------|-------|
|a      |0      |x      |0      |x      |
|b      |x      |0      |x      |0      |
|c      |x      |1      |x      |1      |
|d      |1      |x      |1      |x      |

Ausgabedatei "ZMnichtoptimiert.tbl" erfolgreich geschrieben
Ausgabedatei "ZMoptimiert.tbl" erfolgreich geschrieben
```

Hier zeigt die Optimierung erste Auswirkungen, die Anzahl der Gatter zur Realisierung konnte deutlich reduziert werden.

Tabelle 3: Optimierte Codierung

Zustände	Codierung
b	00
a	01
d	11
c	10

```
1 Gatteranzahl von '../../zmnichtoptimiert.min': 15
2 Gatter:
3 D00 = (D ^ C ^ C01') v (C' ^ C00) v (C00 ^ C01')
4 D01 = (D' ^ C ^ C01) v (C ^ C00' ^ C01) v (D' ^ C ^ C00') v (C' ^ C00)
5 Gatteranzahl von '../../zmoptimiert.min': 10
6 Gatter:
7 D00 = (D ^ C ^ C00' ^ C01) v (C00 ^ C01') v (C' ^ C00)
8 D01 = (D ^ C00 ^ C01) v (C')
```

5.3 Großes Beispiel

Als großer Automat wurde ein 8-Bit Mustererkenner für drei Muster mit 15 Zuständen verwendet. Die erkannten Muster sind `01100110`, `10100110` und `10100111`.

```
Anzahl Zustaende: 15
Anzahl Eingangssignale: 1
Anzahl Ausgangssignale: 1
Reihenfolge Eingangssignale: E
Reihenfolge Ausgangssignale: A

|       |0      |1      |
|-------|-------|-------|
|S0     |S1     |S2     |
|S1     |S3     |S4     |
|S2     |S4     |S3     |
|S3     |S7     |S7     |
|S4     |S7     |S9     |
|S7     |S10    |S10    |
|S11    |S12    |S13    |
|S9     |S11    |S10    |
|S10    |S13    |S13    |
|S12    |S14    |S15    |
|S13    |S14    |S14    |
|S14    |S17    |S17    |
|S15    |S17    |S16    |
|S16    |S0     |S0     |
|S17    |S0     |S0     |
|-------|-------|-------|
|S0     |0      |0      |
|S1     |0      |0      |
|S2     |0      |0      |
|S3     |0      |0      |
|S4     |0      |0      |
|S7     |0      |0      |
|S11    |0      |0      |
|S9     |0      |0      |
|S10    |0      |0      |
|S12    |0      |0      |
|S13    |0      |0      |
|S14    |0      |0      |
|S15    |0      |0      |
|S16    |1      |1      |
|S17    |0      |0      |
```

Tabelle 4: Optimierte Codierung

Zustände	Codierung
S0	0000
S16	0001
S17	0011
S15	0010
S14	0110
S12	0111
S13	0101
S11	0100
S10	1100
S9	1101
S7	1111
S3	1110
S4	1010
S1	1011
S2	1001

Bei 15 Zuständen zeigt sich das volle Potenzial des Optimierungsalgorithmus, durch die optimierte Zustandscodierung kann die Gatteranzahl mehr als halbiert werden.

```
1  Gatteranzahl von '../../zmnichtoptimiert.min': 74
2  Gatter:
3  D00 = (C00' ^ C01 ^ C02' ^ C03) v (E' ^ C00' ^ C01 ^ C02 ^ C03')
4        v (E ^ C00' ^ C01 ^ C02) v (E' ^ C00 ^ C01 ^ C02' ^ C03')
5        v (E ^ C00 ^ C01 ^ C02' ^ C03') v (C00 ^ C01')
6  D01 = (E' ^ C00 ^ C01 ^ C02' ^ C03') v (E ^ C00 ^ C01 ^ C02' ^ C03')
7        v (E' ^ C00' ^ C01' ^ C02) v (E' ^ C01 ^ C02 ^ C03)
8        v (C00' ^ C01 ^ C02' ^ C03') v (E ^ C01' ^ C03) v (C00 ^ C02 ^ C03)
9  D02 = (E' ^ C00 ^ C01 ^ C02' ^ C03') v (E' ^ C01 ^ C02 ^ C03)
10       v (E' ^ C01' ^ C02' ^ C03) v (E ^ C00' ^ C03')
11       v (C00 ^ C01' ^ C03') v (C00 ^ C02 ^ C03)
12 D03 = (E' ^ C00' ^ C01 ^ C02 ^ C03') v (E ^ C00 ^ C01 ^ C02' ^ C03')
13      v (E' ^ C00' ^ C01' ^ C02') v (E ^ C01' ^ C02 ^ C03')
14      v (C00 ^ C01' ^ C02 ^ C03') v (C00' ^ C01' ^ C02 ^ C03)
15      v (E' ^ C01' ^ C02' ^ C03) v (C00' ^ C01 ^ C02' ^ C03')
16 Gatteranzahl von '../../zmoptimiert.min': 36
17 Gatter:
18 D00 = (E ^ C00 ^ C02' ^ C03) v (C01' ^ C02' ^ C03') v (C00 ^ C01' ^ C03)
19       v (C00 ^ C01 ^ C02) v (C00 ^ C01' ^ C03')
20 D01 = (E ^ C00 ^ C02' ^ C03) v (E' ^ C00 ^ C01' ^ C02) v (E' ^ C01 ^ C03)
21       v (C00 ^ C01 ^ C02) v (C00 ^ C01' ^ C03') v (C01 ^ C02')
22 D02 = (E' ^ C00 ^ C01' ^ C02) v (E' ^ C00' ^ C03') v (C00' ^ C01 ^ C03)
23       v (C01 ^ C02 ^ C03') v (C00 ^ C01' ^ C03)
24 D03 = (C03')
```