

**Projektbericht**

**Algorithmen und Datenstrukturen**

**Prof.Dr.-Ing. Schäfer**

**Hazardüberprüfung in zweistufigen Gatternetzen**

**Teilnehmer:**

**Gael Dongmo Kouakam (32410)**

**Jonathan Stump (xxxx)**

**Wintersemester 2013/2014**

Inhaltverzeichnis

[I. Projektbeschreibung 3](#_Toc377838796)

[II. Beschreibung des Programmes 4](#_Toc377838807)

[II.1. Klasse PrimImplikantCollection: 4](#_Toc377838808)

[II.2. Klasse Cparser: 5](#_Toc377838815)

[II.3. Klasse PrimImplikant: 5](#_Toc377838817)

[II.4. Klasse Cell: 6](#_Toc377838819)

[II.7. Klasse Tools: 12](#_Toc377838835)

[II.8. Klasse KV\_PiELeLoc: 13](#_Toc377838837)

[II.9. Klasse KV\_PiGroup: 14](#_Toc377838839)

[II.10. Klasse KV: 15](#_Toc377838843)

[III. PAP des Programmes 23](#_Toc377838879)

[IV. Quellen 24](#_Toc377838880)

# I. Projektbeschreibung

Bei der Realisierung von asynchronen rückgekoppelten Schaltwerken kann eine mögliche Fehlfunktion durch die Überprüfung der im Karnaugh-Diagramm eingezeichneten Implikanten gefunden werden. Hierzu sind benachbarte Gruppen zu suchen, die horizontal oder vertikal in einem Schritt einen Übergang zwischen beiden Gruppen erlauben. Werden die Implikanten in der Form P1(a,b,c,d)=110x und P2(a,b,c,d)=1x11 geschrieben, so besteht ein Unterschied nur an einer Stelle. An Stellen in den ein x vorkommt, muss kein Überprüfung

vorgenommen werden. Im vorliegenden Fall wären P1 und P2 benachbart, da nur an der dritten Stelle ein Unterschied zu verzeichnen ist. Entsprechende Hazardterme zur Vermeidung von Signaleinbrüchen sind dann an diesen Stellen zu ermitteln. Die Untersuchungen sollen

sowohl für disjunktive als auch für konjunktive Normalformen vorgenommen werden können.

**Schnittstellenbeschreibung:**

**Eingabedaten:**

Variables: a, b, c, d

Terms:

“1000”>1 //Angabe in binärer Form. Erlaubt 0,1,x

6>1 //Angabe in dezimaler Form, x kann hier nicht kodiert werden.

**Ausgabe:**

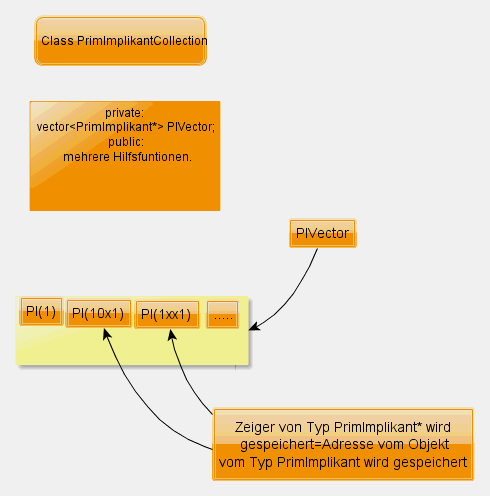
Die Ausgabe soll sowohl wahlweise textuell in Form einer Wertetabelle oder booleschen Gleichungen als auch in Form eines Karnaugh-Diagramms ausgegeben werden können.

**Hilfsmittel:**

Programm lexan Kapitel 7.3 Skript Algorithmen und Datenstrukturen Asynchrone Schaltwerke Kapitel 1.5 Skript Digitale Systeme GDE – Software.

**II. Beschreibung des Programmes**

**II.1. Klasse PrimImplikantCollection:**

****

Sie speichert Zeiger von Typ PrimImplikant in seinem Vector (PIVEctor) anhand überladener Funktionen:

void add(string input); // der PrimImplikant kann einen String seien.

void add(uint input); // der PrimImplikant kann direkt einen Integer seien.

void add(uint input1, uint input2); // der PrimImplikant hat 2 Interger Werte.

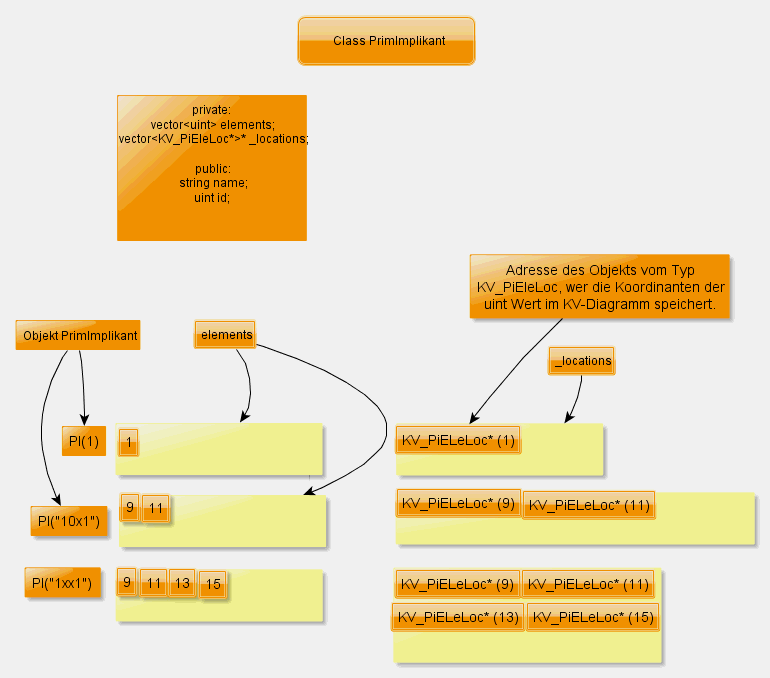
Es wird innerhalb diesen oben genannten Methoden erstens einen dynamischen Zeiger vom Typ PrimImplikant mit dem String oder dem Interger als Argumenten allokiert. Zweitens wird dieser Zeiger(PrimImplikant\*) im PiVector anhand der Funktion „void

add(PrimImplikant\* &PI);“

**II.2. Klasse Cparser:**

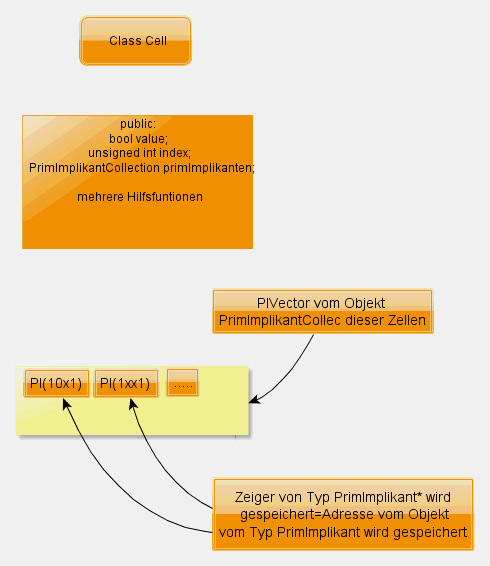
Die hat als Hauptziel die in der Textdatei(input.txt) liegenden Daten herauszufiltern. Sie ist eigentlich einen „Lexical Parser“(lexikalischer Analysator). Die Funktion **yyparser**  besitzt als Argumenten einen Zeiger(globalPIC) von Typ PrimImplikantCollection und einen anderen Zeiger(variables) von Typ vector<string>. Während dem Filtern werden die PrimImplikanten(String oder Interger) mit einer der überladenen Funktion abgefangen. Die Variablen werden in variables gespeichert.

**II.3. Klasse PrimImplikant:**

****

Die Methode **parser** bekommt nur den PrimImplikant von dem Typ String und generiert alle seine möglichen binären Zahlen, falls ein „x“ im String vorhanden ist. Diese werden im Vektor Elements gespeichert. Die Koordinaten (w;h)von jedem Element sind wichtig. Jeder Element besitzt eine Kästchen im KV-Diagramm, die der Breite(w)und der Tiefe(h) entspricht. Der Vektor **\_locations** speichert jede einzelne Instanz der Klasse KV\_PiELeLoc mit in ihr die gespeicherte Elementskoordinanten.

**II.4. Klasse Cell:**



Eine Zelle im KV-Diagramm beinhalten eine Wert (0 oder 1), ein Index(Ganzzahl) und seine PrimImplikanten im Vektor, in denen ihre Index auftritt. Die Methode **getNeighbors**  speichert in einem Vektor alle Nachbarnzellen. So sieht den Algorithmus aus:

vector<Cell\*>\* Cell::getNeighbors(vector<Cell\*> &allCells)

{

vector<Cell\*>\* neighbors = new vector<Cell\*>();

neighbors->resize(dimension);

uint j = 1;

for (unsigned char i = 0; i < dimension; i++) // for all bits

{

neighbors->at(i) = allCells[this->index ^ j]; // add Cell that

//is different at one bit (j)

j <<= 1;

}

return neighbors;

}

Es gibt die Methode **getHazards**. Die prüft, ob die die Value der im Vektor neighbors gespeicherten Zellen „1 “ sind. Danach wird die Methode **hasOneofThose** die jeweilige PrimImplikantvektoren mit dem der zentralen Zelle überprüft. Falls sie mindestens eine gemeinsame PrimImplikant haben, gibt es keinen Hazard. Ansonsten tritt einen Hazard auf und muss das entsprechende Objektadresse vom Typ Cell im Vektor Hazardous gespeichert.

vector<Cell\*>\* Cell::getHazards(vector<Cell\*> &allCells)

{

vector<Cell\*>\* hazardous = new vector<Cell\*>();

vector<Cell\*>\* neighbors = this->getNeighbors(allCells);

for (vector<Cell\*>::iterator neighbor = neighbors->begin(); neighbor < neighbors->end(); neighbor++) // for each neighbor

{

if ((\*neighbor)->value == false) // if the neighbor is

//not covered by a PrimImpliaknt, no hazard can occur

continue;

if ((\*neighbor)->hasOneOfThose(this->primImplikanten) == false) // check whether at least one PrimImplikant that covers this Cell also //covers the neighbor cell

hazardous->push\_back(\*neighbor); // if not, add it to hazardous

}

delete neighbors;

return hazardous;

}

bool Cell::hasOneOfThose(PrimImplikantCollection &foreignPic)

{

for (uint i = 0; i < foreignPic.size(); i++)// for each foreign PrimImplikant

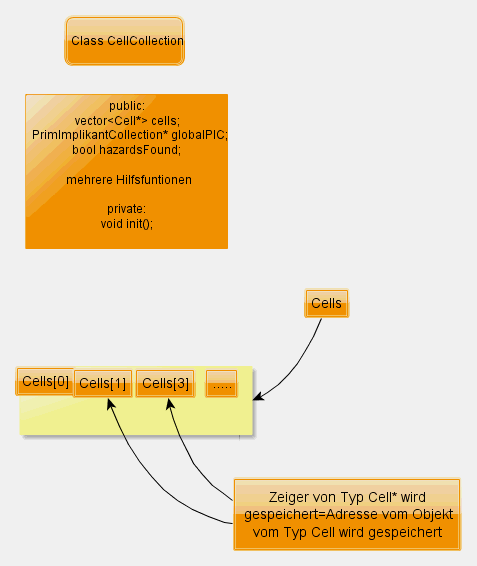
if (this->primImplikanten.contains(foreignPic[i])) // check whether it also covers this Cell

return true;

return false;

}

**II.5. Klasse CellCollection:**

****

Der Vektor Cells wird mit der Länge 2^(Anzahl der Variables) initialisiert.

Der Zeiger globalPIC ist notwendig, um gewisse Eigenschaften der Objekten von Typ Cell zu initialisieren. Die Methode **FindHazards** wandelt die Ganzzahl in Gray, indem man EXOR benuzt: **i(gray)=i(d)^i(d)/2.** Das muss gemacht werden, weil die Reihenfolge der Zellen im KV Diagramm nach Graycodierung ist. Um die Geschwindigkeit bei der Suche der Hazards (mit getNeighbors und getHarzards )zu verdoppeln , Wird das Schachmuster eingesezt, indem nur jede Zweite entfernte Zelle von der kontrollierte Zelle auch geprüft wird. Das liegt daran, dass die Methode getNeighbors schon die benachbarten Zellen gecheckt hat. Und es wäre umständlich noch die direkte Nachbarzelle (I(gray)++) mit den alten Nachbarzelle(i(gray)) zu checken. Falls Hazards existieren, muss eine neue PrimImplikant mit Ihren Indexzahlen(in Gray) im Pivector generiert werden. Die PrimImplikantvektoren dieser HazardsZellen werden updatet.

**II.6. Klasse Wertetabelle:**

Da wir alle Daten haben, wird eine Ausgabe der Wahrheitstabelle sowohl auf dem Fenster als auch in der Textdatei **wertetabelle.txt** vorgenommen. Folgende sind Eigenschaften und Methoden:

public:

void Print(); // print the complete truth table

/// <summary>

/// Constructor of Wertetabelle

/// </summary>

Wertetabelle(CellCollection\* cells, vector<string>\* variables, ofstream &fWt)

{

this->cells = cells; // store a reference to allCells (thus we will get all changes)

this->variables = variables;

this->fot = &fWt;

}

private:

void makeHeader(); // Generates the header lines

void printHeader(); // Prints the header lines

void printI(uint i); // Prints the binary representation of the specified i in table format

void printPrimImplikanten(uint i); // Print all PrimImplikants that cover the specified position i

CellCollection\* cells; // all cells

vector<string>\* variables; // variable names

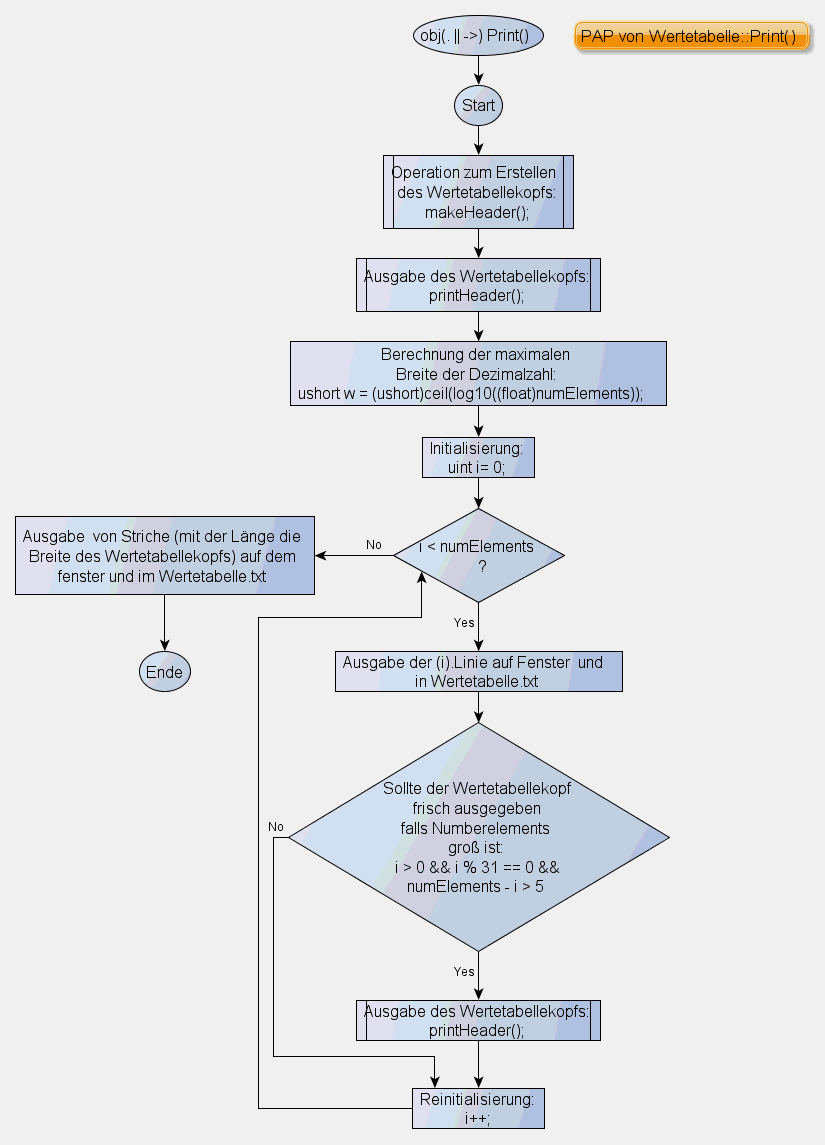
vector<float> padding;// padding depending on length of variable names

uint width; // with of truth table

ofstream\* fot; // output file stream

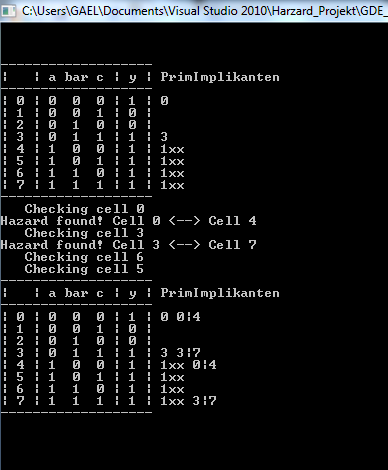
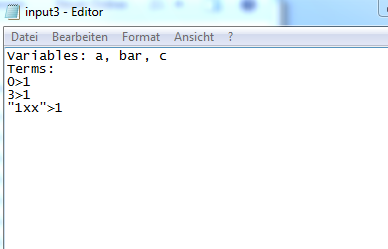
string header; // header line

Einen Überblick über die Methode **Print():**

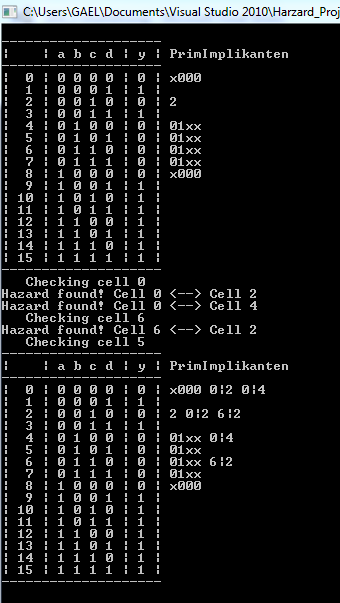
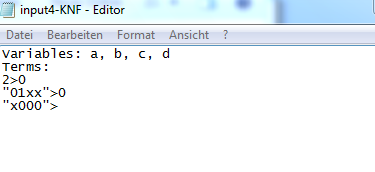


Einen Einblick über die Ausgabe am Fenster:

* **Dimension:3-----🡪 (a,bar,c)**

* **Dimension: 4 -----**🡪**(a,b,c,d)**

****

**II.7. Klasse Tools:**

class Tools

{

public:

static char\* Tools::BinaryToChars(uint x, char length); // convert a number to a string in it's binary representation

static uint GrayToBinary(uint x); // convert a gray number back to binary using a lookup table

static bool compareAsGray(uint a, uint b);// compary towo gray numbers (convert it first)

private:

static uint\* GrayToBinaryTable; // gray to binary lookup table

static uint GrayToBinaryTableSize; // size of GrayToBinaryTable

static void InitGrayToBinaryTable(); // init GrayToBinaryTable

static uint Tools::ConvertGrayToBinary(uint x); // convert a gray number back to binary

};

Eine seiner Methoden wird in der Klasse Wertetabelle benutzt. Nämlich char\* Tools::BinaryToChars(uint x, char length) um die Ganzzahle in binär darzustellen. Die restlichen Methoden( GrayToBinary(uint x), compareAsGray(uint a, uint b), ConvertGrayToBinary(uint x), InitGrayToBinaryTable())erlauben eine aufwärts Aufstellung der Objekteadressen der Klasse **KV\_PiEleLoc** im Vektor **\_locations.** Das führt dazu, dass die Eigenschaft **Index**(Gray) von jedem Objekt nach der Aufwärts Aufstellung von Dezimalzahlen eingereiht ist. Es ist hilfsreich um die Kullers eines PrimImplikant zu bestimmen.

**II.8. Klasse KV\_PiELeLoc:**

class KV\_PiEleLoc

{

public:

uint i;

uint w;

uint h;

/// <summary>

/// Constructor of KV\_PiEleLoc

/// </summary>

KV\_PiEleLoc(uint &i)

{

const uint numVarX = (uint)floor(dimension / 2.0f); // get the number of variables in X direction

this->i = i;// we need numVarX ones here

// example: numVarX = 3 --> (0001 << 3) - 1 = 1000 - 1 = 0111

this->w = i & ((0x1 << (numVarX)) - 1); // get x coord out of i (lower numVarX bits)

this->w = Tools::GrayToBinary(w)// convert from gray to binary

this->h = i >> numVarX; // get y coord out of i (higher bits)

this->h = Tools::GrayToBinary(h);// convert from gray to binary

}

Dank dem Wert i(Gray) und die auf w und h vorgenommenen Ausmarkierungen werden bzw. die Breite und der Tiefe ermittelt. Danach werden sie in binärzahl um gewandelt. Diese erlauben später in der Methode **KV::Print()** die exakten Koordinaten von jedem Element(Gray) zu berechnen.

**II.9. Klasse KV\_PiGroup:**

class KV\_PiGroup

{

private:

vector<KV\_PiEleLoc\*> elements; // locations

public:

bool LiesNextTo(KV\_PiEleLoc\* &el); // whether the specified el lies next to this group

void Add(KV\_PiEleLoc\* &el); // add a location to this group

void MakeCoords(uint edgeLength, uint VarX\_Length, uint VarY\_Length, uint\* X1, uint\* Y1, uint\* X2, uint\* Y2); // Generates the coordinates for this group

KV\_PiEleLoc\* operator[](uint &index);// location at index

KV\_PiEleLoc\* at(uint &index); // location at index

uint size(); // number of locations

};

Jeder PrimImplikant kann mehrere Kullers haben. Deswegen brauchen wir diese Klasse, die die zu dieser Klasse (auch als Kuller betrachtet) gehörten Elemente speichert. Die Methode **LiesNextTo** überprüft, ob die nächste Element des Vektors \_locations in der Nähe ist. Wenn Ja wird diese Element in der internen Vektor dieser Klasse gespeichert.Im dem Algorithmus dieser Methode wird die Differenz der Breite u. Tiefe zw. den zu überprüfenden Element mit denen von dem internen Vektor(**elements**). Am Ende der Überprüfung ist mit der Methode **MakeCoords** die 4 Koordinanten zum Zeichnen des Kullers im KV Diagramm bestimmt.

bool KV\_PiGroup::LiesNextTo(KV\_PiEleLoc\* &el)

{

for (uint i = 0; i < this->elements.size(); i++) // for each element in this group

{ // (there should be a better way)

KV\_PiEleLoc\* elG = this->elements[i];

if (elG->h - el->h == 0 && abs((int)elG->w - (int)el->w) == 1 || // if el lies next to elG in X direction

elG->w - el->w == 0 && abs((int)elG->h - (int)el->h) == 1) // or el lies next to elG in Y direction

return true;

}

return false;

}

void KV\_PiGroup::MakeCoords(uint edgeLength, uint VarX\_Length, uint VarY\_Length, uint\* XX1, uint\* YY1, uint\* XX2, uint\* YY2)

{

uint x1, x2, y1, y2; // coords for each single element

uint X1 = -1, X2 = 0, Y1 = -1, Y2 = 0; // resulting group coords with initial maximum/minimum values

for (uint i = 0; i < this->elements.size(); i++) // for all locations in this group

{

KV\_PiEleLoc\* loc = this->elements[i]; // current location

x1 = loc->w \* (edgeLength + 1) + VarY\_Length; // Upper coord for rectangle around loc

x2 = x1 + edgeLength; // Lower coord for rectangle around loc

y1 = loc->h \* (edgeLength + 1) + VarX\_Length; // Left coord for rectangle around loc

y2 = y1 + edgeLength; // Right coord for rectangle around loc

X1 = min(X1, x1); // Store it if upper coord is higher than previous X1

X2 = max(X2, x2); // Store it if lower coord is lower than previous X2

Y1 = min(Y1, y1); // Store it if left coord is lefter than previous Y1

Y2 = max(Y2, y2); // Store it if right coord is righter than previous Y2

}

\*XX1 = X1;

\*YY1 = Y1;

\*XX2 = X2;

\*YY2 = Y2;

}

**II.10. Klasse KV:**

In dieser Klasse wird es gezeichnet und ausgefüllt. Und zwar die Variablen, die Werte 0 oder 1 in der Zelle, die Kullers und die Grayzahle.

Funktionen zum Zeichen im GDE wurde im KV Klasse unter Methoden importiert:

void Line(uint x1, uint y1, uint x2, uint y2, int color); // Zeichnet eine Linie mit Offset

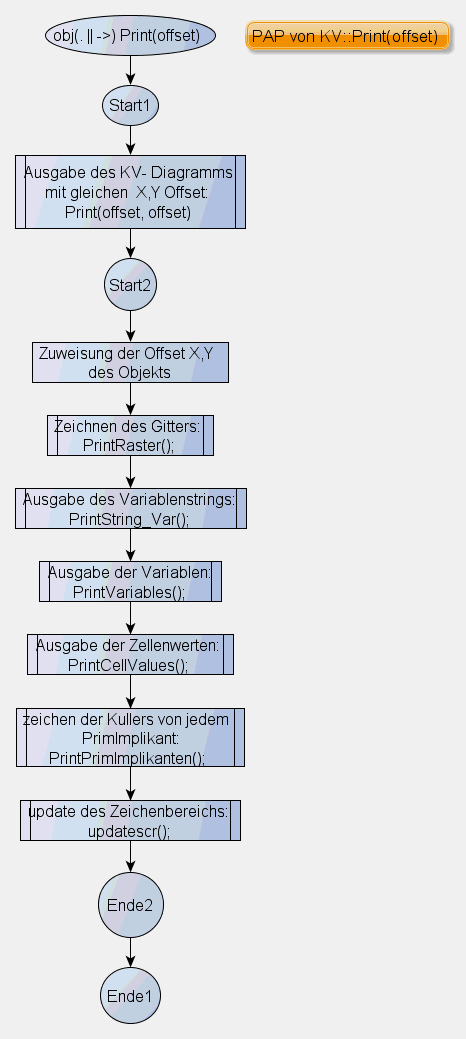
void Text(uint x, uint y, uint size, int color, int bkcolor, int angle, char\* theText); // Zeichnet einen Text mit Offset

void KV::TextBox(uint x1, uint y1, uint x2, uint y2, uint size, int ctext, int cframe, int cfill, int flags, char\* theText); // Zeichnet eine TextBox mit Offset

void KV::TextBoxBold(uint x1, uint y1, uint x2, uint y2, uint size, int ctext, int cframe, int cfill, int flags, char\* theText); // Zeichnet eine TextBox mit Offset und fetter Schrift

void KV::Rectangle(uint x1, uint y1, uint x2, uint y2, int cframe, int cfill); // Zeichnet ein Rechteck mit Offset

Folgende andere wichtige Methoden von KV im **KV::Print()** und einen Einblick der Ausgabe mit Z.B. 3 Variables :

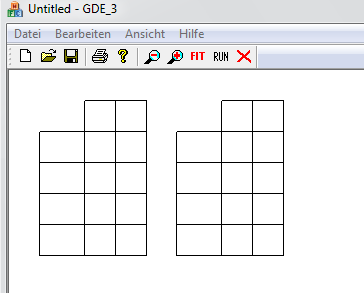


**Schritt1:**

* void PrintRaster(); // Erstellt die Felder/das Raster

Links: KV Diagramm ohne Hazardsüberprüfung

Rechts: KV Diagramm mit Hazardsüberprüfung

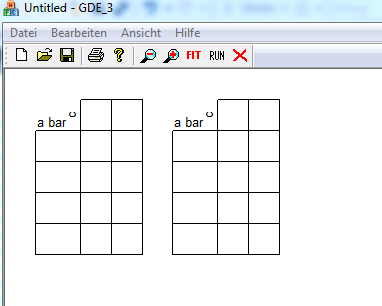


**Schritt 2:**

* void PrintRaster(); // Erstellt die Felder/das Raster
* void PrintString\_Var();// Erstellt den horizontalen & vertikalen Text mit Variablennamen

Links: KV Diagramm ohne Hazardsüberprüfung

Rechts: KV Diagramm mit Hazardsüberprüfung

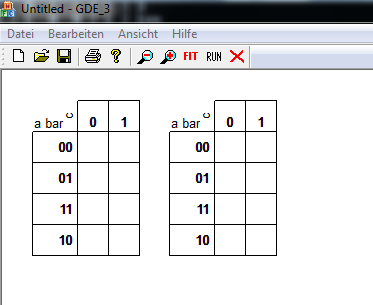


**Schritt 3:**

* void PrintRaster(); // Erstellt die Felder/das Raster
* void PrintString\_Var();// Erstellt den horizontalen & vertikalen Text mit Variablennamen
* void PrintVariables();// Erstellt die Werte der Variablen in der ersten Zeile und der ersten Spalte

Links: KV Diagramm ohne Hazardsüberprüfung

Rechts: KV Diagramm mit Hazardsüberprüfung

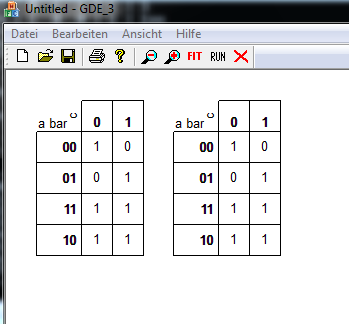


**Schritt 4:**

* void PrintRaster(); // Erstellt die Felder/das Raster
* void PrintString\_Var();// Erstellt den horizontalen & vertikalen Text mit Variablennamen
* void PrintVariables();// Erstellt die Werte der Variablen in der ersten Zeile und der ersten Spalte
* void PrintCellValues();// Erstellt die Werte in den jeweiligen Zellen

Links: KV Diagramm ohne Hazardsüberprüfung

Rechts: KV Diagramm mit Hazardsüberprüfung

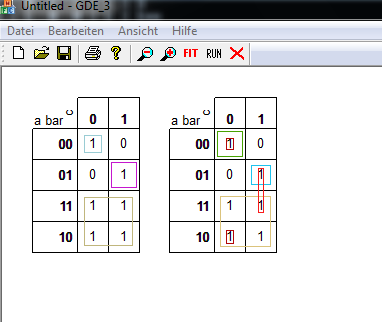
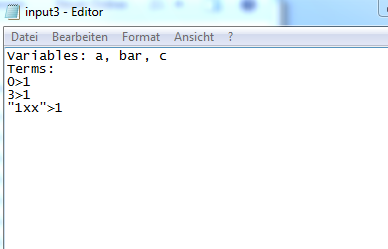


**Schritt 5:**

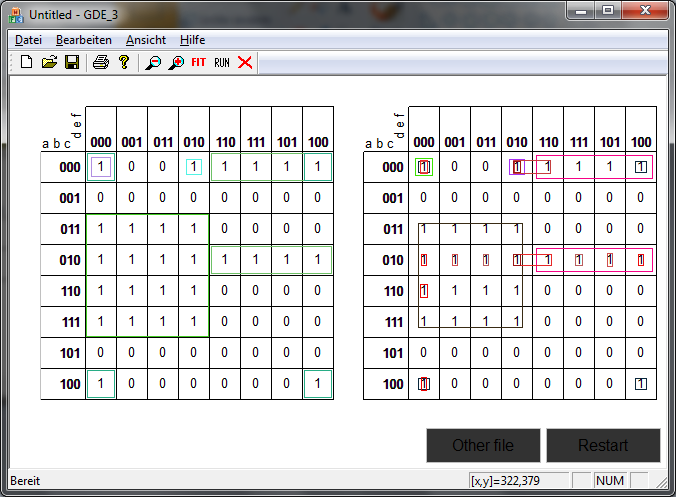
* void PrintRaster(); // Erstellt die Felder/das Raster
* void PrintString\_Var();// Erstellt den horizontalen & vertikalen Text mit Variablennamen
* void PrintVariables();// Erstellt die Werte der Variablen in der ersten Zeile und der ersten Spalte
* void PrintCellValues();// Erstellt die Werte in den jeweiligen Zellen
* void PrintPrimImplikanten();// Generiert die einzelnen PrimImplikanten-Kuller (Gruppen)

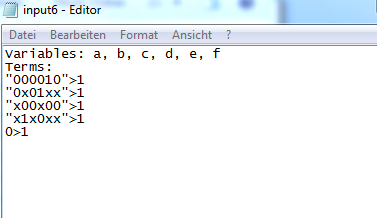
Links: KV Diagramm ohne Hazardsüberprüfung

Rechts: KV Diagramm mit Hazardsüberprüfung

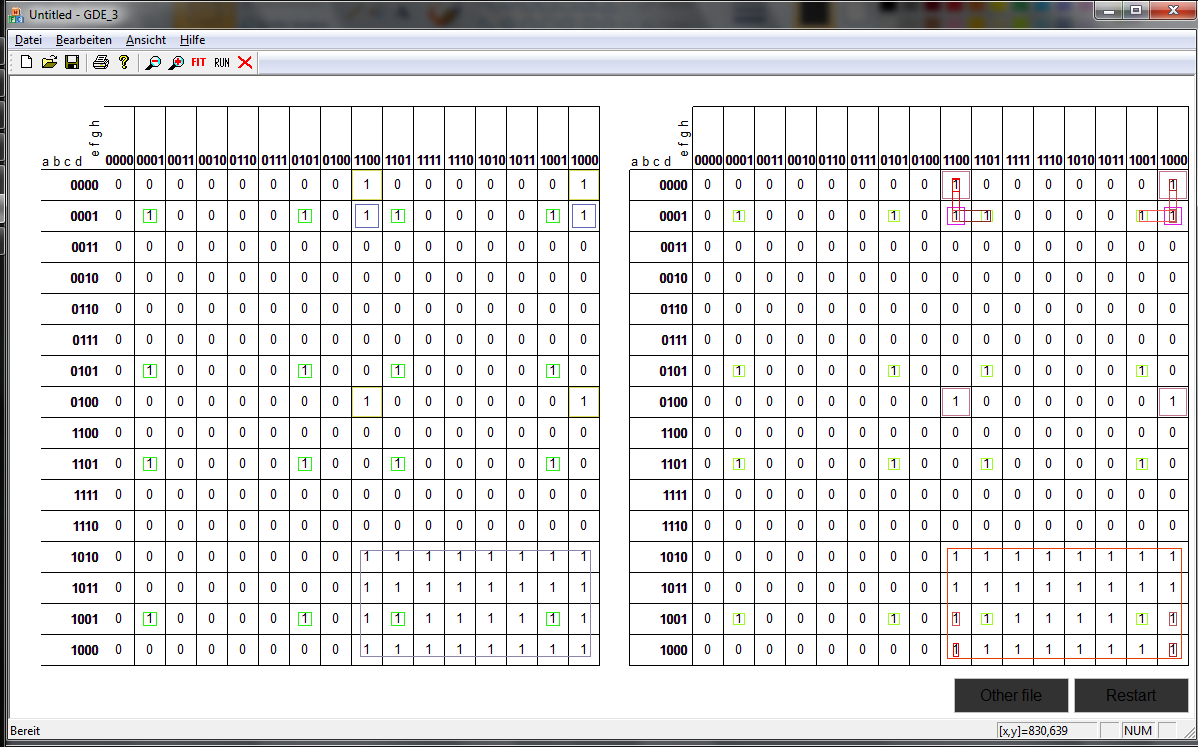


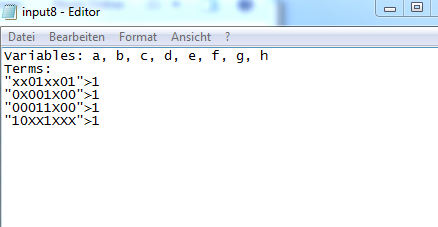
**Ausgabe mit 6 Variablen**:



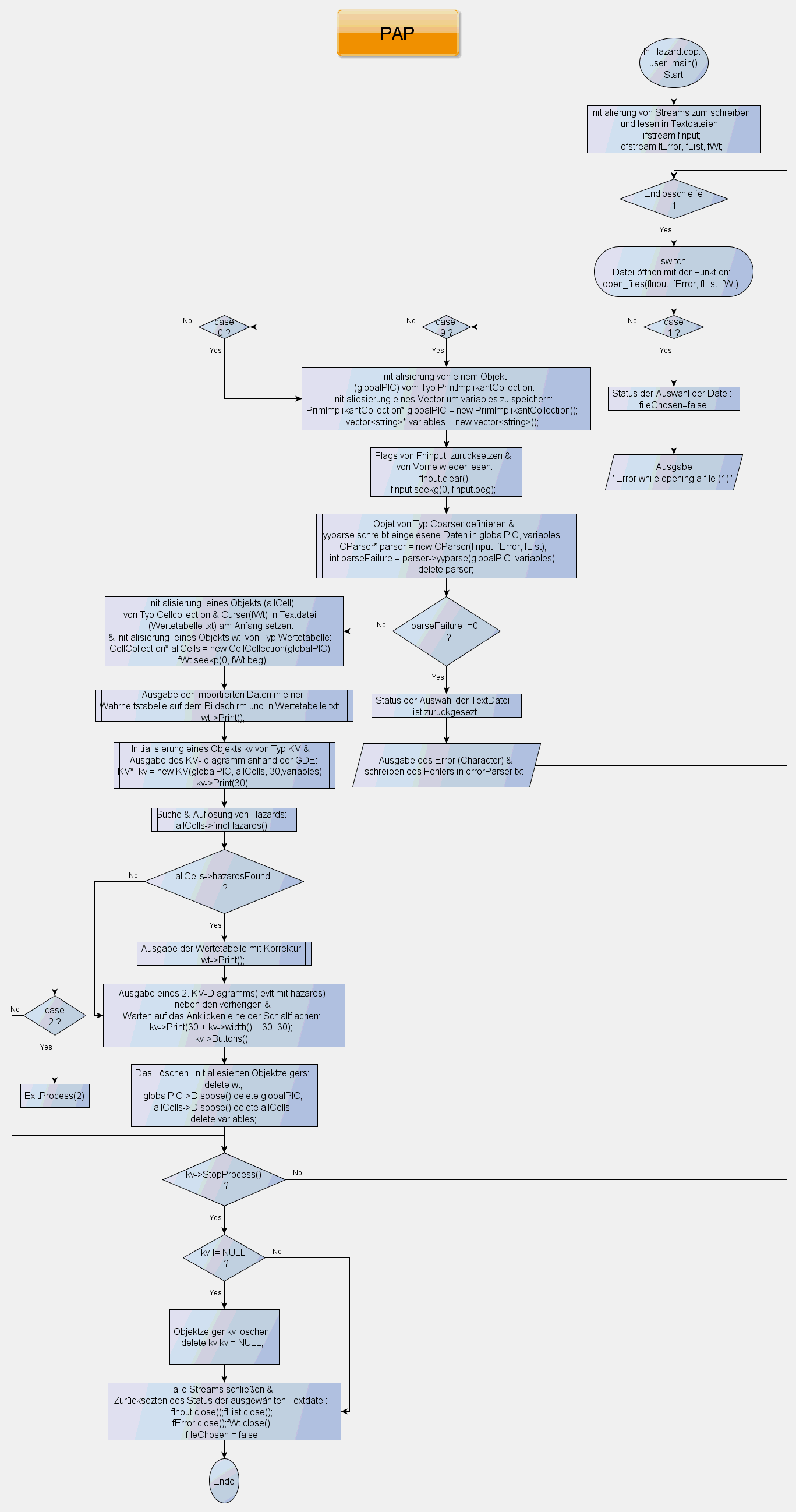


**Ausgabe mit 8 Variablen:**

****



**III. PAP des Programmes**



**IV. Quellen**

* http://www.c-plusplus.de/forum