

**Projektbericht**

**Algorithmen und Datenstrukturen**

**Prof. Dr.-Ing. Schäfer**

**Hazard-Überprüfung in zweistufigen Gatternetzen**

**Teilnehmer:**

**Gael Dongmo Kouakam (32410)**

**Jonathan Stump (35852)**

**Wintersemester 2013/2014**

Inhaltsverzeichnis

[1 Projektbeschreibung 3](#_Toc377930166)

[2 Beschreibung des Programmes 4](#_Toc377930167)

[2.1 Klasse PrimImplikant 4](#_Toc377930168)

[2.2 Klasse PrimImplikantCollection 5](#_Toc377930169)

[2.3 Klasse CParser 5](#_Toc377930170)

[2.4 Klasse Cell 6](#_Toc377930171)

[2.5 Klasse CellCollection 8](#_Toc377930172)

[2.6 Klasse Wertetabelle 9](#_Toc377930173)

[2.7 Klasse Tools 12](#_Toc377930174)

[2.8 Klasse KV\_PiELeLoc 13](#_Toc377930175)

[2.9 Klasse KV\_PiGroup 13](#_Toc377930176)

[2.10 Klasse KV 15](#_Toc377930177)

[3 PAP der Hauptfunktion 20](#_Toc377930178)

[4 Quellen 21](#_Toc377930179)

# 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 denen ein x vorkommt, muss keine Überprüfung vorgenommen werden. Im vorliegenden Fall wären P1 und P2 benachbart, da nur an der dritten Stelle (c) ein Unterschied zu verzeichnen ist. Entsprechende Hazard-Terme 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 nicht kodiert werden.

**Ausgabe:**

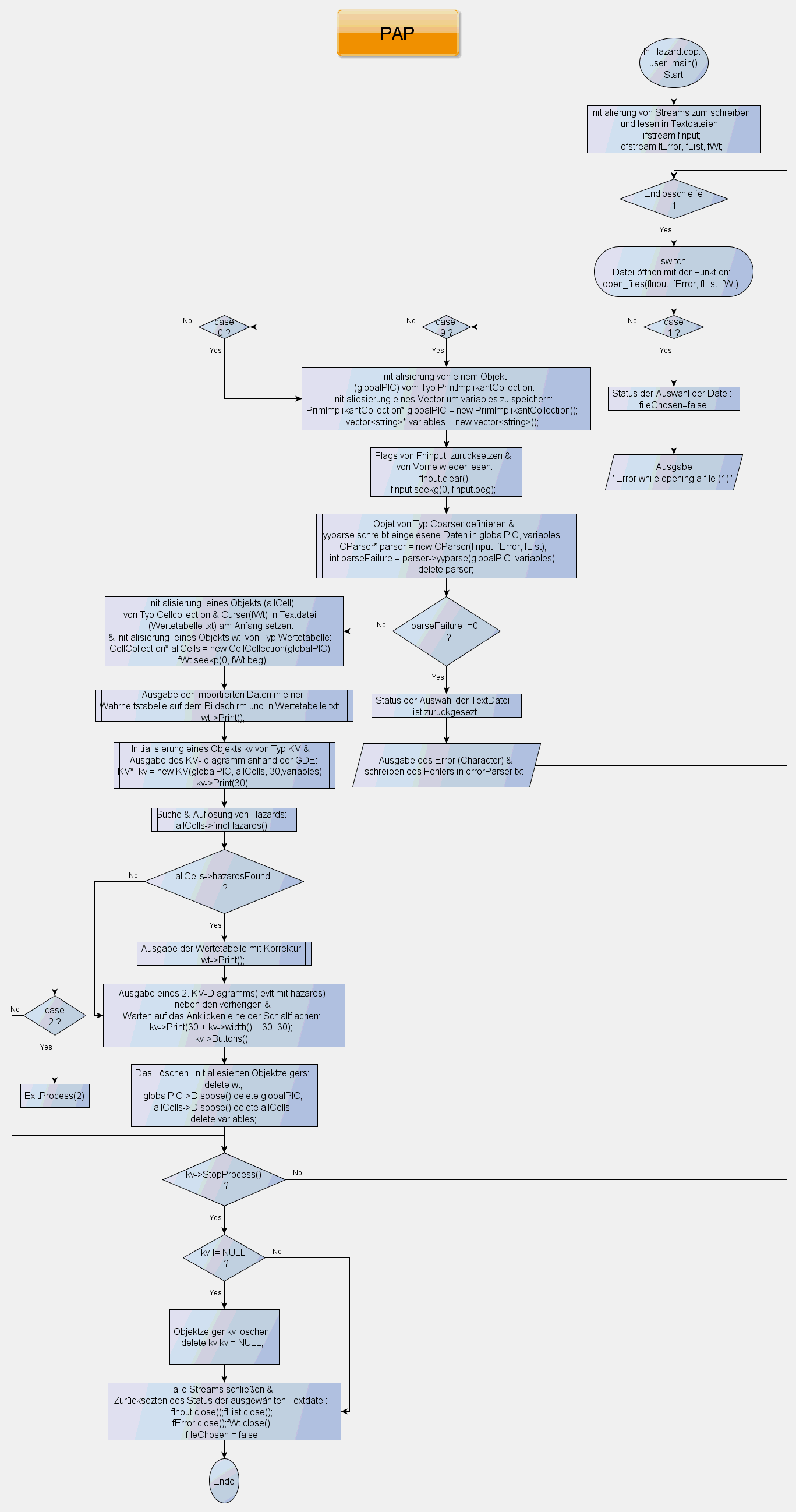
Die Ausgabe soll sowohl wahlweise textuell in Form einer Wertetabelle oder booleschen Gleichungen als auch in Form eines Karnaugh-Diagramms ausgegeben werden können.

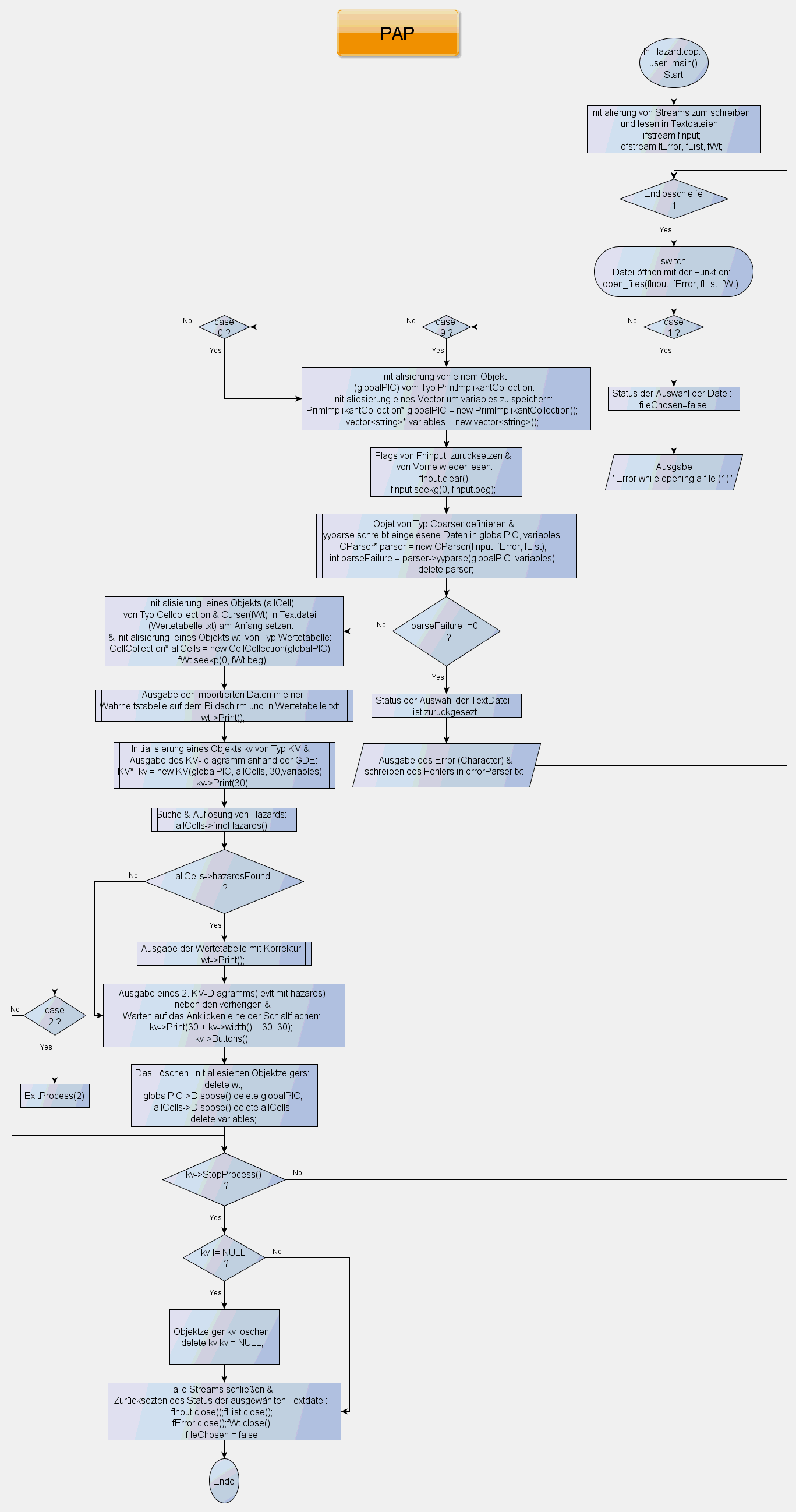
**Hilfsmittel:**

* Programm „lexan“, Kapitel 7.3 Skript Algorithmen und Datenstrukturen
* Asynchrone Schaltwerke, Kapitel 1.5 Skript Digitale Systeme
* GDE – Software

# Beschreibung des Programmes

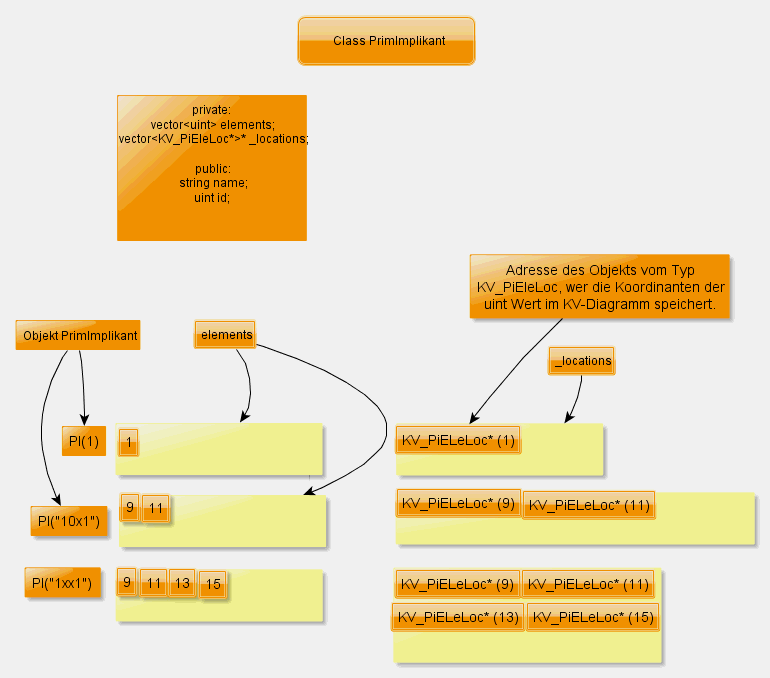
## PAP der Hauptfunktion





## Klasse PrimImplikant

Diese Klasse stellt den grundsätzlichsten Datentyp in unserer Applikation dar und repräsentiert einen logischen Primimplikanten.

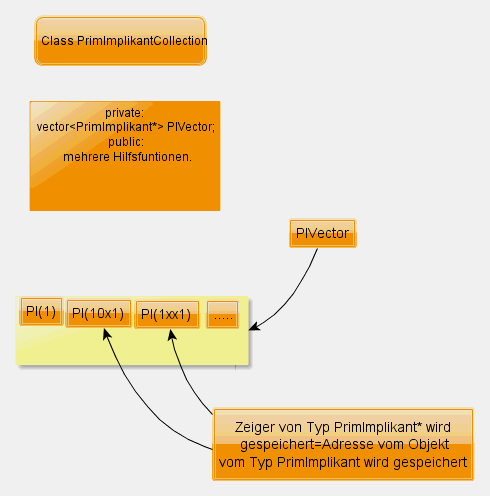


Die Aufgabe Konstruktors dieser Klasse ist, die von der Klasse CParser (s. Kapitel 2.3) eingelesenen Daten in eine oder mehrere Integer umzuwandeln und diese dem Vector elements hinzuzufügen. Von ihm wird die Methode parser() wird benutzt, um einen eingelesenen String zu verarbeiten. Sie ersetzt ein gefundenes „x“ sowohl durch eine „0“ als auch durch eine „1“ und ruft sich jeweils erneut auf. Schlussendlich, wenn kein „x“ mehr gefunden wurde, wird der String in eine Zahl umgewandelt und abgespeichert.

Für die grafische Ausgabe im KV-Diagramm sind die Koordinaten (x; y) eines jeden Elements wichtig. Jede Integer besitzt ein Kästchen im KV-Diagramm, das mit den Nummern der Spalte (x) und der Zeile (y) lokalisiert werden kann. Der Vektor \_locations speichert für jedes Element eine Instanz der Klasse KV\_PiELeLoc mit den gespeicherten Koordinanten.

## Klasse PrimImplikantCollection

Die Klasse PrimImplikantCollection speichert Zeiger auf Objekte vom Typ [PrimImplikant](#_Klasse_PrimImplikant) (s. Kapitel 2.1) in einem Vector (PIVector) anhand überladener Funktionen und stellt weitere Methoden bereit, um diese Menge an Objekten zu verarbeiten.



void add(string input); // PrimImplikant kann ein String sein.

void add(uint input); // PrimImplikant kann direkt ein Integer sein.

void add(uint input1, uint input2); // PrimImplikant hat 2 Integer-Werte.

Innerhalb dieser Methoden wird zuerst ein Objekt vom Typ [PrimImplikant](#_Klasse_PrimImplikant) angelegt und der eingelesene String oder die Integer dem Konstruktor als Argument übergeben. Anschließend wird der Zeiger auf das neue Objekt dem PiVector hinzugefügt.

Zusätzlich werden noch folgende Methoden bereitgestellt:

// returns the PrimImplikanten that touch the specified position

PrimImplikantCollection primImplikantenAt(uint position);

// looks for any PrimImplikant that touches the specified position

bool valueAt(uint position);

// looks for the specified PrimImplikant in this colleciton

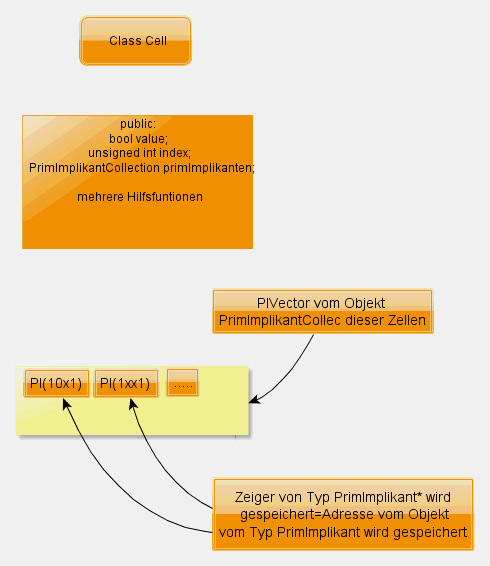
bool contains(PrimImplikant\* foreign);

## Klasse CParser

Das Hauptziel der Klasse CParser ist es, die in der Eingabetextdatei (input.txt) liegenden Daten einzulesen und zu analysieren. Ihr Hauptbestandteil ist ein „Lexical Parser“ (lexikalischer Analysator). Der Hauptfunktion yyparser() werden als Argumente ein Zeiger auf ein Objekt vom Typ [PrimImplikantCollection](#_Klasse_PrimImplikantCollection) (globalPIC, welche alle [PrimImplikant](#_Klasse_PrimImplikant)en enthalten wird) und ein Zeiger auf ein Objekt vom Typ vector<string> übergeben. Während des Einlesens werden die [PrimImplikant](#_Klasse_PrimImplikant)en (String oder Integer) der add()-Funktion der [PrimImplikantCollection](#_Klasse_PrimImplikantCollection) übergeben und die angegebenen Variablennamen im entsprechenden Vector abgespeichert.

## Klasse Cell

Um die [PrimImplikant](#_Klasse_PrimImplikant)en strukturiert verarbeiten zu können, ist eine Klasse von Nöten, die einen bestimmten Zustand der Variablen repräsentiert. In Anlehnung an das KV-Diagramm heißt diese Klasse „Cell“.



Eine Zelle im KV-Diagramm wird durch den entsprechenden Zustand seiner Variablen eindeutig identifiziert, welcher als Integer abgespeichert werden kann. Darüber hinaus bestimmt die eingelesene kombinatorische Schaltung den Ausgangswert zu diesem Zustand (0 oder 1).

Um die [PrimImplikant](#_Klasse_PrimImplikant)en effektiv verarbeiten zu können, wird für jede Zelle ein [PrimImplikantCollection](#_Klasse_PrimImplikantCollection)-Objekt erzeugt, das diejenigen [PrimImplikant](#_Klasse_PrimImplikant)en enthält, die diese Zelle betreffen.

Die Methode getNeighbors() gibt alle benachbarten Zellen, also diejenigen Variablenzustände mit der Hammingdistanz 1 zum aktuellen Zustand, in einem Vector zurück:

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

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

neighbors->resize(dimension); // number of defined variables

uint j = 1;

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

{

// add Cell that is different at one bit (j)

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

j <<= 1;

}

return neighbors;

}

Die Methode getHazards() prüft anhand der oben generierten Liste, ob eine der benachbarten Zelle von einem [PrimImplikant](#_Klasse_PrimImplikant)en bedeckt wird, ohne dass sich die beiden [Zellen](#_Klasse_Cell) einen [PrimImplikant](#_Klasse_PrimImplikant)en teilen (dies ist die Definition eines Hazards, s. Kapitel 0). Hierzu wird die Methode *hasOneofThose()* verwendet, die die beiden [PrimImplikantCollection](#_Klasse_PrimImplikantCollection)s miteinander vergleicht. Falls ein Hazard auftritt, wird die entsprechende Objektadresse der Nachbarzelle im Vektor hazardous gespeichert, welcher am Ende zurückgegeben wird.

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

{

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

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

for (vector<Cell\*>::iterator nb = neighbors->begin();

nb < neighbors->end(); nb++) // for each neighbor

{

// if neighbor is not covered by a PrimImpliaknt

if ((\*nb)->value == false) continue; // no hazard can occur

// check whether at least one PrimImplikant that covers

// this Cell also covers the neighbor cell

if ((\*nb)->hasOneOfThose(this->primImplikanten) == false) hazardous->push\_back(\*nb); // if not, add it }

delete neighbors;

return hazardous;

}

bool Cell::hasOneOfThose(PrimImplikantCollection &foreignPic)

{

// for each foreign PrimImplikant

for (uint i = 0; i < foreignPic.size(); i++)

// check whether it also covers this Cell

if (this->primImplikanten.contains(foreignPic[i]))

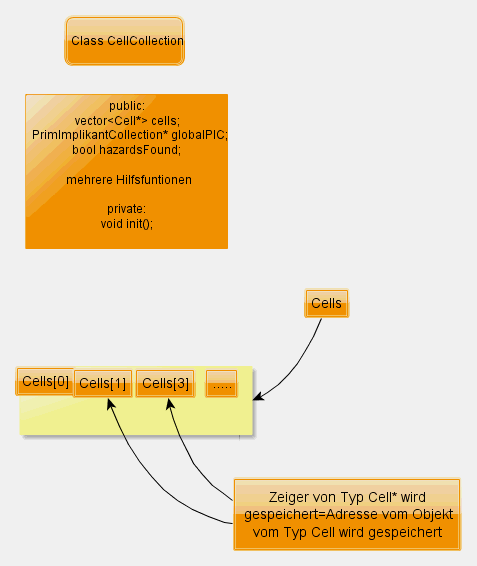
return true;

return false; // no PrimImplikant matched

}

## Klasse CellCollection

Die Klasse CellCollection enthält einen Vektor mit Variablenzuständen ([Cell](#_Klasse_Cell)s, s. Kapitel 2.4) und stellt Methoden zur Verfügung, die diese bearbeiten. In der Applikation gibt es eine Instanz, die alle verfügbaren [Cell](#_Klasse_Cell)s speichert.



Der dem Konstruktor übergebene Zeiger globalPIC wird verwendet, um die [Zellen](#_Klasse_Cell) zu erstellen und deren [PrimImplikantCollections](#_Klasse_PrimImplikantCollection) zu initialisieren und später auch zu aktualisieren.

Die Methode FindHazards() implementiert den Algorithmus, der alle Variablenzustände durchläuft und nach Hazards durchsucht (s. Cell::getHazards() in Kapitel 2.4).

Um die Laufzeit des Algorithmus zu verbessern, kann bei Enumeration der Zellen mit Hammingdistanz 1 jeder zweite Variablenzustand übersprungen werden, da ein Hazard nicht gerichtet ist. Die Hazard-Prüfung von Zelle A nach Zelle B ist somit identisch mit der Prüfung von Zelle B nach Zelle A. Um besagte Enumeration zu erhalten, wird eine binäre Zahl stetig um zwei inkrementiert und anschließend mit XOR in Gray umgewandelt:

igray = id ^ (id/2)

Bei Markierung der behandelten Zellen mit dem Index igray ergibt sich auf dem KV-Diagramm ein Schachbrettmuster. Darüber hinaus können Zellen, die von keinem [PrimImplikant](#_Klasse_PrimImplikant)en abgedeckt werden, übersprungen werden.

Sobald ein Hazard detektiert wurde, wird ein neuer [PrimImplikant](#_Klasse_PrimImplikant) mit den Indizes der beiden Zellen generiert und der globalen [PrimImplikantCollection](#_Klasse_PrimImplikantCollection) hinzugefügt. Um die Änderungen in die Zellen zu übernehmen, werden deren [PrimImplikantCollection](#_Klasse_PrimImplikantCollection) aktualisiert.

## Klasse Wertetabelle

Diese Klasse ist dafür verantwortlich, eine Wertetabelle in der Konsole als auch in einer Textdatei (**res\wertetabelle.txt)** auszugeben. Folgende sind ihre Eigenschaften und Methoden:

public:

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

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

private:

void makeHeader(); // Generates the header lines

void printHeader(); // Prints the header lines

void printI(uint i); // Prints the binary representation of i

void printPrimImplikanten(uint i); // Print all PIs that cover Cell i

CellCollection\* cells; // all cells

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

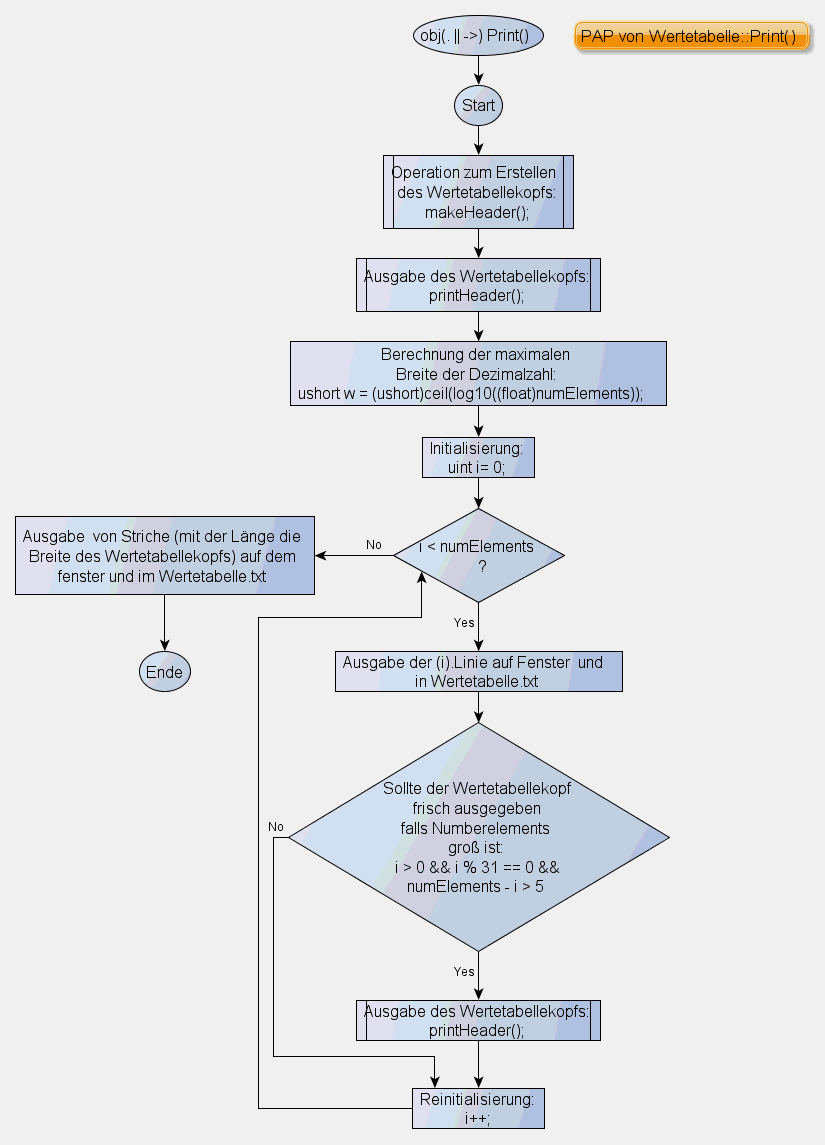
vector<float> padding; // padding depending on length of variable names

uint width; // with of truth table

ofstream\* fot; // output file stream

string header; // header line

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



Ein Einblick in die Ausgabe in der Konsole:

|  |  |
| --- | --- |
| **Dimension: 3 (Variablennamen: a, bar, c)** | |
|  |  |
| **Dimension: 4 (Variablennamen: a, b, c, d)** | |
|  |  |

## Klasse Tools

Die Klasse Tools ist vollständig statisch und stellt Methoden zur Verfügung, die von verschiedenen Klassen genutzt warden können. So wird BinaryToChars() von [Wertetabelle](#_Klasse_Wertetabelle) (s. Kapitel 2.6) benutzt, um Zahlen in ihrer binären Form in einen String mit einer festen Länge umzuwandeln.

Die restlichen Methoden werden dazu verwendet, eine gray-codierte Zahl in ihre normale Form umzuwandeln, wofür mehrere Gray-Codierungen notwendig sind.

class Tools

{

public:

// convert a number to a string in its binary representation

static char\* Tools::BinaryToChars(uint x, char length);

// convert a gray number back to binary using a lookup table

static uint GrayToBinary(uint x);

// compary two gray numbers (convert it first)

static bool compareAsGray(uint a, uint b);

private:

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

static uint GrayToBinaryTableSize; // size of GrayToBinaryTable

static void InitGrayToBinaryTable(); // init GrayToBinaryTable

// convert a gray number back to binary

static uint Tools::ConvertGrayToBinary(uint x);

};

## Klasse KV\_PiELeLoc

Diese Klasse speichert lediglich die X- und Y-Koordinaten der Zelle im KV-Diagramm.

class KV\_PiEleLoc

{

public:

uint i;

uint w;

uint h;

KV\_PiEleLoc(uint &i)

{

// get the number of variables in X direction

const uint numVarX = (uint)floor(dimension / 2.0f);

this->i = i;

// get x coord out of i (lower numVarX bits)

// we need numVarX ones here

// z.B.: numVarX = 3 --> (0001 << 3) - 1 = 1000 - 1 = 0111

this->w = i & ((0x1 << (numVarX)) - 1);

this->w = Tools::GrayToBinary(w); // decode

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

this->h = Tools::GrayToBinary(h); // decode

}

};

Aus dem Index iGray der Zelle werden die zwei Werte w und h ausmaskiert. Anschließend werden sie in normale Binärzahl umgewandelt, um Spalte und Zeile der Zelle im KV-Diagramm zu erhalten.

## Klasse KV\_PiGroup

In einem KV-Diagramm besitzen alle Nachbarn einer Zelle die Hammingdistanz 1 zu der zentralen Zelle. Da sich in jede Dimension zwei Nachbarzellen auftragen lassen (z.B. links und rechts oder oben und unten), ergibt sich bei mehr als vier Eingangsvariablen ein Problem. Die Ausgabe auf dem Bildschirm ist auf zwei Dimensionen beschränkt, bei höherer Variablenanzahl liegen nicht mehr alle Zellen neben den Zellen, deren Variablenkombination sich nur in einem Bit von ihnen unterscheidet.

Somit kann jeder [PrimImplikant](#_Klasse_PrimImplikant) aus mehreren „Kullern“ bestehen. Die Klasse KV\_PiGroup stellt einen solchen Kuller dar, indem sie eine Gruppe von [KV\_PiEleLocs](#_Klasse_KV_PiELeLoc) (s. Kapitel 2.8) speichert.

Die Methode LiesNextTo()überprüft, ob ein Objekt vom Typ [KV\_PiEleLoc](#_Klasse_KV_PiELeLoc) am Rand einer solchen Gruppe liegt und somit zu dieser hinzugefügt werden kann.

Um die Klasse [KV](#_Klasse_KV) (s. Kapitel 2.10) zu unterstützen, generiert diese Klasse in der Methode MakeCoords()die X- und Y-Koordinaten zum Zeichnen des entsprechenden Rechtecks im KV-Diagramm.

class KV\_PiGroup

{

private:

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

public:

// whether the specified el lies next to this group

bool LiesNextTo(KV\_PiEleLoc\* &el);

// add a location to this group

void Add(KV\_PiEleLoc\* &el);

// Generates the coordinates for this group

void MakeCoords(uint edgeLength, uint VarX\_Length,

uint VarY\_Length, uint\* X1, uint\* Y1, uint\* X2, uint\* Y2);

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

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

uint size(); // number of locations

}

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

{

// for each element in this group

for (uint i = 0; i < this->elements.size(); i++)

{

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

if (

// if el lies next to elG in X direction

elG->h - el->h == 0 &&

abs((int)elG->w - (int)el->w) == 1 ||

// or el lies next to elG in Y direction

elG->w - el->w == 0 &&

abs((int)elG->h - (int)el->h) == 1)

return true;

}

return false;

}

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

{

// coords for each single element

uint x1, x2, y1, y2;

// resulting group coords with initial maximum/minimum values

uint X1 = -1, X2 = 0, Y1 = -1, Y2 = 0;

// for all locations in this group

for (uint i = 0; i < this->elements.size(); i++)

{

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

// Upper coord for rectangle around loc

x1 = loc->w \* (edgeLength + 1) + VarY\_Length;

// Lower coord for rectangle around loc

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

y1 = loc->h \* (edgeLength + 1) + VarX\_Length;

// Right coord for rectangle around loc

y2 = y1 + edgeLength;

// Store it if upper coord is higher than previous X1

X1 = min(X1, x1);

// Store it if lower coord is lower than previous X2

X2 = max(X2, x2);

// Store it if left coord is lefter than previous Y1

Y1 = min(Y1, y1);

// Store it if right coord is righter than previous Y2

Y2 = max(Y2, y2);

}

\*XX1 = X1; \*YY1 = Y1; \*XX2 = X2; \*YY2 = Y2;

}

## Klasse KV

In dieser Klasse wird das KV-Diagramm gezeichnet und gefüllt. Ein KV-Diagramm besteht aus dem Grundgerüst, den Variablennamen, den Zellwerten (0 oder 1) sowie den [PrimImplikant](#_Klasse_PrimImplikant)en.

Die GDE-Zeichen-Funktionen wurden in die KV-Klasse inkludiert, um das Zeichnen mehrer KV-Diagramme durch unterschiedliche Offsets zu vereinfachen:

// Zeichnet eine Linie mit Offset

void Line(uint x1, uint y1, uint x2, uint y2, int color);

// Zeichnet einen Text mit Offset

void Text(uint x, uint y, uint size, int color, int bkcolor, int angle, char\* theText);

// Zeichnet eine TextBox 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);

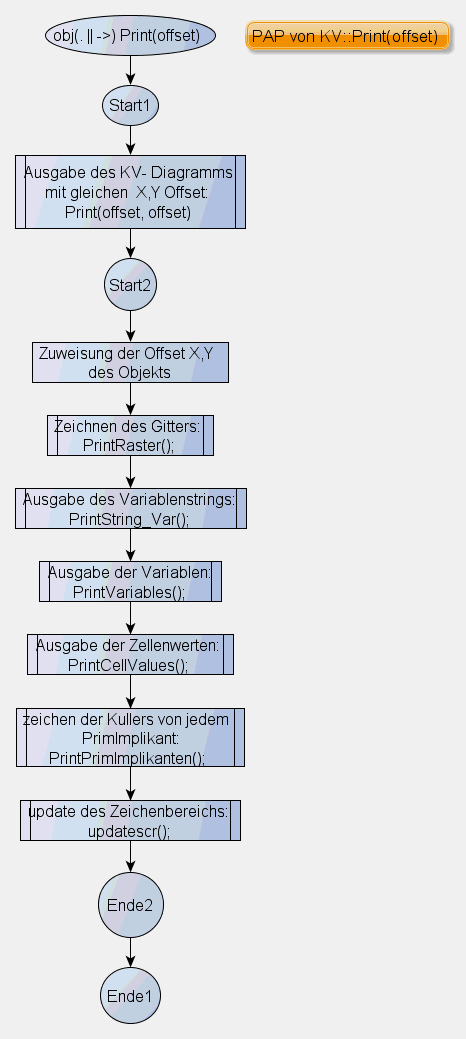
// Zeichnet eine TextBox mit Offset und fetter Schrift

void KV::TextBoxBold(uint x1, uint y1, uint x2, uint y2, uint size, int ctext, int cframe, int cfill, int flags, char\* theText);

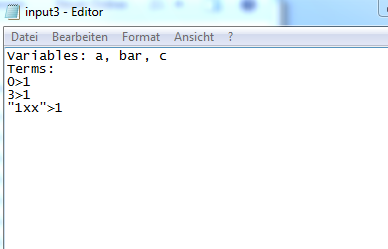
// Zeichnet ein Rechteck mit Offset

void KV::Rectangle(uint x1, uint y1, uint x2, uint y2, int cframe, int cfill);

Folgende Methoden sind im Ablauf von KV::Print() ebenfalls von Bedeutung und werden beispielhaft für 3 Variablen visualisiert:

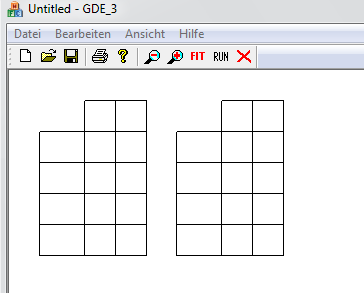


Eingangsdaten:



* **Schritt1**

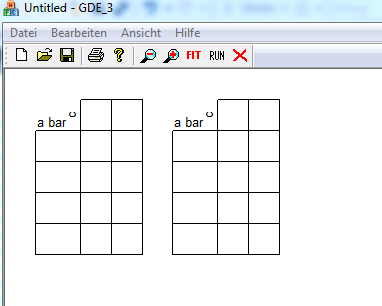
void PrintRaster(); // Erstellt die Felder/das Raster



* **Schritt 2**

void PrintString\_Var();

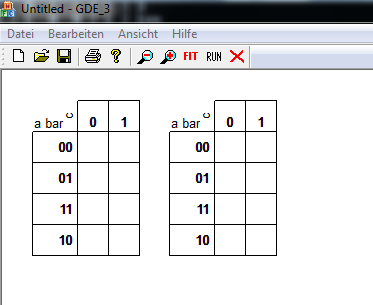
// Erstellt den horizontalen & vertikalen Text mit Variablennamen



* **Schritt 3**

void PrintVariables();

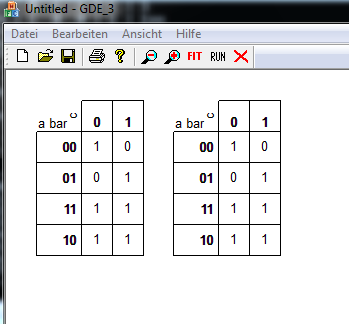
// Erstellt die Variablen in der 1. Zeile und der 1. Spalte



* **Schritt 4**

void PrintCellValues();

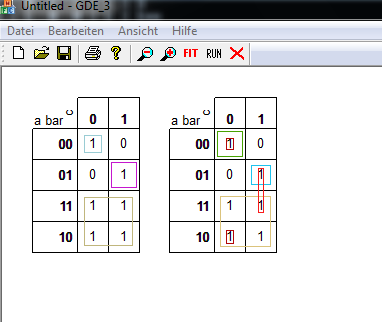
// Erstellt die Werte in den jeweiligen Zellen



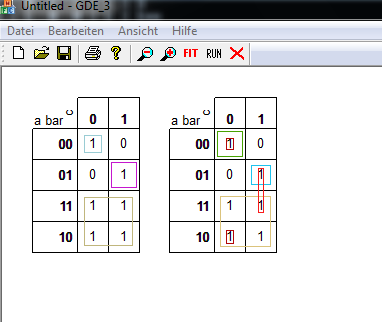
* **Schritt 5**

void PrintPrimImplikanten();

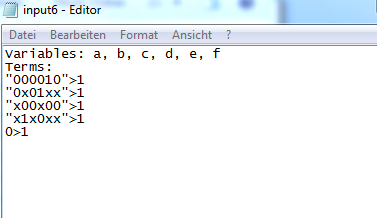
// Generiert die einzelnen PrimImplikanten-Kuller (-Gruppen)

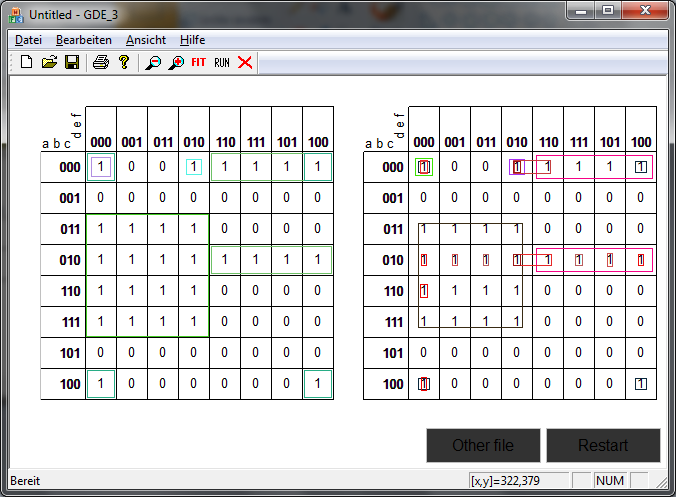


Durch mehrmaliges Aufrufen mit unterschiedlichen Offsets können mehrere KV-Diagramme nebeneinander gezeichnet werden, z.B. um den Zustand vor und nach der Detektion der Hazards zu verdeutlichen:

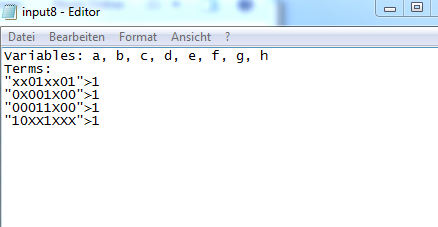


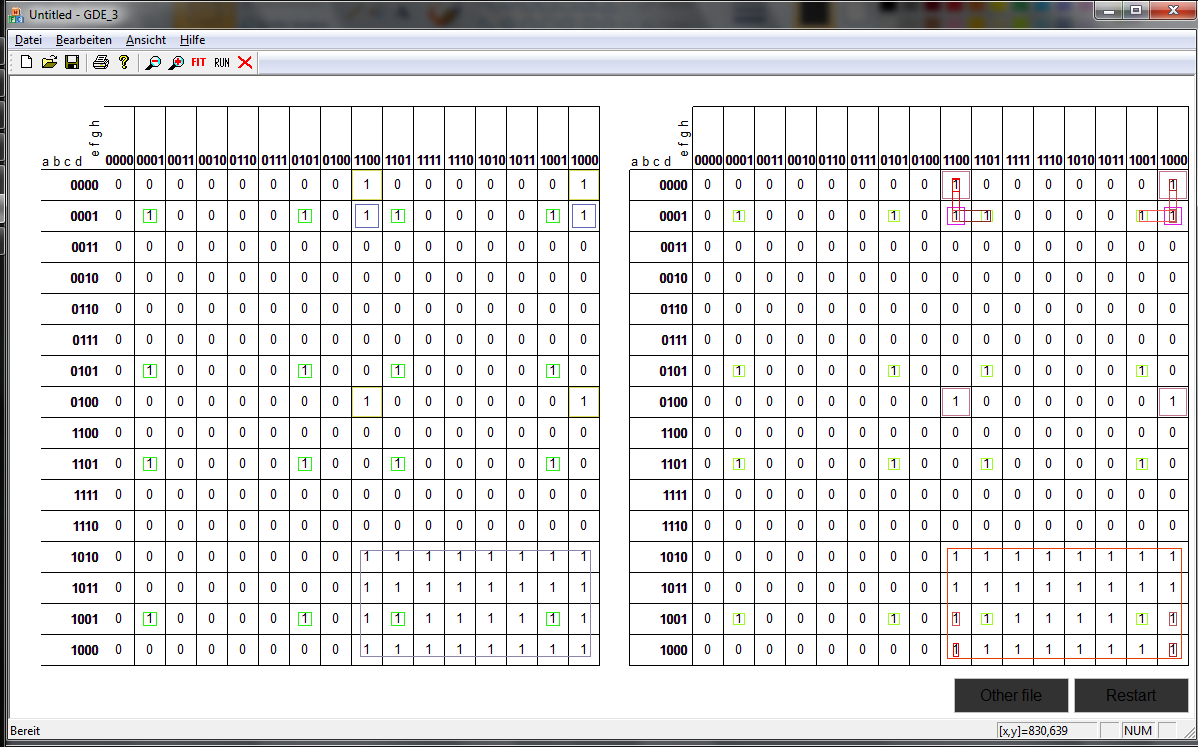
**KV-Diagramm mit 6 Variablen:**





**KV-Diagramm mit 8 Variablen:**





# Quellen

* <http://www.cplusplus.com/>
* <http://www.c-plusplus.de/forum>