Diplomarbeit

Automatenbasierte Entscheidungsverfahren für Presburger Arithmetik

Christoph Wagner
März 2006

Betreuer:
Prof. Dr. rer. nat. Klaus Schneider
Dipl. - Inf. Tobias Schüle
Arbeitsgruppe Reaktive Systeme
Fachbereich Informatik
Universität Kaiserslautern

Die vorliegende Diplomarbeit wurde von mir selbständig verfasst. Es wurden keine anderen als die angegebenen Quellen und Hilfsmittel benutzt.

Kaiserslautern, den 28. März 2006

Christoph Wagner

Inhaltsverzeichnis

1 Einführung

Der Einsatz reaktiver Systeme in sicherheitskritischen Umgebungen hat in den letzten Jahren dazu geführt dass viel an Methoden der Verifikation von reaktiven Systemen gearbeitet wurde. Dabei hat der Einsatz von symbolischen Repräsentationsformen für die zu untersuchenden Systeme die Effizienz der Verifikation deutlich verbessert. Die Grundlagen der symbolischen Repräsentation liegen in der Verwendung von binären Entscheidungsdiagrammen (BDDs) als eine kanonische Normalform für aussagenlogische Formeln. Die Modellierung mit BDDs hat aber den Nachteil, dass nicht-boolsche Variablen, wie Integer nur eine bestimmte Länge n aufweisen können und sich dadurch der Zustandsraum um 2^n vergrößert.
Eine andere Modellierungsmethode ist die Verwendung von Presburger Arithmetik. Diese ermöglicht die Verwendung von positiven ganzen Zahlen und die Addition darauf. Die Arithmetik kann aber auf Integer erweitert werden, und behält dabei die Eigenschaft vollständig und entscheidbar zu sein.
Für die Presburger Arithmetik existiert auch eine symbolische Darstellung. Es können Automaten konstruiert werden, die Wörter akzeptieren welche Variablenbelegungen entsprechen, die die Formel erfüllen. Die Verwendung von Automaten bildet wiederum eine kanonische Normalform und wird daher häufig verwendet.
Die Entscheidbarkeit der Formel läßt sich bei Automaten auf die Prüfung der Leerheit des Automaten zurückführen. Denn akzeptiert der Automat kein Wort, so gibt es kein Modell, welches die zugehörige Formel erfüllt.
Die Konstruktion von Automaten weist aber nicht elementare Laufzeit auf. Für jede auftretende Quantorenalternation in der zu übersetzenden Formel wird eine Determinisierung des konstruierten Automaten nötig, welche eine Laufzeit von 2^n benötigt.
In der vorliegenden Arbeit wird eine bisher noch nicht näher untersuchte Automatenklasse betrachtet und auf ihre Eignung zur Entscheidung von Presburger Arithmetik untersucht.
Die betrachteten Signed-Digit-Automaten basieren auf Signed-Digit-Zahlen und weisen durch deren Eigenschaften eine ganz eigene Struktur auf. Es wird betrachtet, wie sich Presburger Arithmetik in diese Automaten übersetzen lässt, und mit welchen Methoden diese Automaten auf Leerheit untersucht werden können.
Es wird gezeigt wie diese Automaten implementiert werden können, und wie dabei BDDs benutzt werden können, um Mengen von Signed-Digits symbolisch darzustellen. Die Implementierung wird dann anhand von gemessenen Zeiten für die Übersetzung und die Entscheidung bewertet.

2 Voraussetzungen

2.1 Signed-Digit-Zahlen

Die sogenannten Signed-Digit-Zahlen sind Zahlen zu einer Basis B die so erweitert werden dass jede Ziffer ein Vorzeichen besitzt.

Definition 1 (Signed-Digit-Zahl). *Sei zu einer gegebenen Basis $B \in \mathbb{N}$ ein A gegeben mit $0 < A < B$. Dann ist $\mathcal{D} = \{-A, \ldots, A\}$ die Menge der Signed-Digit-Ziffern. Eine Signed-Digit (kurz SD) Zahl $x = [x_{n-1}, \ldots, x_0]$ ist eine Folge von Ziffern mit $x_i \in \mathcal{D}$, $0 \leq i < n$.*

Definition 2 (Wert einer SD-Zahl). *Der Wert einer SD-Zahl $x = [x_{n-1}, \ldots, x_0]$ in einer Basis B bestimmt sich zu $\langle [x_{n-1}, \ldots, x_0] \rangle_B := \sum_{i=0}^{n-1} x_i \cdot B^i$*

Definition 3 (Darstellbare Zahlen). *Die Menge der darstellbaren Zahlen einer n - stelligen SD-Zahl ist $\{-A\frac{B^n-1}{B-1}, \ldots A\frac{B^n-1}{B-1}\}$. Die größte darstellbare Zahl ist somit $[A \ldots A]$ und die kleinste $[-A \ldots -A]$*

Definition 4 (Anzahl der Ziffern einer SD-Zahl). *Die minimale Anzahl von Ziffern die benötigt wird um eine Zahl x durch eine SD-Zahl in der Basis B darzustellen läßt sich mit der Funktion $size : \mathbb{Z} \to \mathbb{N}$ mit $size(x) = \lceil log_B(\frac{B-1}{A} \cdot \mid x \mid +1) \rceil$*

Sei im Folgenden $A = B - 1$. Zu bemerken ist, dass SD-Zahlen redundant sind, es also zu einer Zahl x mehrere diese repräsentierende SD-Zahlen gibt.
Als ein Beispiel sei $B = 3$ und $x = 14$. so gibt es unter Anderen die folgenden zwei Repäsentationen: $x_1 = [1, 1, 2]$, $x_2 = [1, 2, -1]$. Die Bestimmung der Werte beider Zahlen zeigt dass beide dieselbe Zahl repräsentieren.
$< x_1 >_B = 1 \cdot 3^2 + 1 \cdot 3^1 + 2 \cdot 3^0 = 14$
$< x_2 >_B = 1 \cdot 3^2 + 2 \cdot 3^1 - 1 \cdot 3^0 = 14$
Ein Sonderfall ist die Darstellung der Zahl 0. Sie ist eindeutig durch $[0, \ldots, 0]$ repräsentiert.

Definition 5 (Komplement einer SD-Zahl). *Zu einer gegebenen SD-Zahl $x = [x_{n-1}, \ldots, x_0]$ ist das Komplement gegeben durch $\bar{x} = [-x_{n-1}, \ldots, -x_0]$*

Definition 6 (Summen- und Übertragsziffer). *Zu zwei SD-Zahlen $x = [x_{n-1}, \ldots, x_0]$ und $y = [y_{n-1}, \ldots, y_0]$ ist der Übertrag gegeben durch*

$$t_i := \begin{cases} +1 & : \textit{falls } x_{i-1} + y_{i-1} \geq A \\ -1 & : \textit{falls } x_{i-1} + y_{i-1} \leq -A \\ 0 & : \textit{sonst} \end{cases}$$

mit $t_0 = 0$ gilt dann, dass $s_i := x_i + y_i + t_i - t_{i+1} \cdot \mathcal{B}$ für $0 < i \leq n$.

Hierbei ist nützlich, dass s_i nur von x_i, y_i, x_{i-1} und y_{i-1} abhängt und somit auch im schlechtesten Fall kein Übertrag von x_0, y_0 bis zu s_i stattfinden kann.

Definition 7 (Addition). *Wird die Basis $B \geq 3$ gewählt, so gilt für die Addition zweier SD-Zahlen x und y dass $\langle [x] \rangle_B + \langle [y] \rangle_B = \langle [t_n, s_{n-1}, \ldots, s_0] \rangle_B$*

Die Subtraktion lässt sich dabei leicht auf die Addition mit dem Komplement des Subtrahenden zurückführen.

2.2 Zweier-Komplement Kodierung

Zweierkomplement Zahlen sind die verbreitetste Kodierung für ganze Zahlen auf binärer Ebene.
Zur Darstellung einer positiven ganzen Zahl x werden $n = \lceil log_2(x+1) \rceil$ Bits benötigt. Damit lassen sich die Zahlen $0 \ldots 2^n - 1$ darstellen. Durch die Erweiterung auf ganze Zahlen, die auch negativ sein können, verdoppelt sich die Menge der darzustellenden Zahlen auf $-2^n - 1 \ldots 2^n - 1$, weshalb n um eins auf $n+1$ erhöht werden muss.
Zur Darstellung einer beliebigen ganzen Zahl x werden also $n = \lceil log_2(x+1) \rceil + 1$ Bits benötigt. Die positiven ganzen Zahlen $0 \ldots 2^{n-1} - 1$ werden wie gewohnt repräsentiert. Die negativen Zahlen werden im Bereich von $2^{n-1} \ldots 2^n - 1$ komplementiert dargestellt.
In Abbildung 1 kann der Wert einer Zweier-Komplement-Zahl eingesehen werden. Diese Darstellung bietet Vorteile im Umgang mit den Zahlen. Das höchstwertige Bit x_{n-1} des Bitvektors X repräsentiert das Vorzeichen. Eine Erweiterung des Bitvektors auf $m > n$ Bits ist leicht, es muss nur das Vorzeichenbit entsprechend vervielfältigt werden. $x_i = x_{n-1}$ für $n < i < m - 1$.
Die Addition zweier beliebiger Zweier-Komplement-Zahlen kann bitweise ohne Beachtung des Vorzeichens vorgenommen werden. Es ist jedoch zu beachten dass ein Übertrag stattfinden kann der den Bitvektor der Summe auf $n+1$ Bit erweitert.
Auch die Komplementbildung fällt leicht. Wenn X der Bitvektor der Zahl x ist, so ist der Bitvektor $\bar{X}$ der Zahl -x gegeben durch: $\bar{X} := [\bar{x}_{n-1}, \ldots, \bar{x}_0] + 1$. Durch das Komplement läßt sich die Subtraktion beliebiger Zweier-Komplement-Zahlen auf die Addition zurückführen.
Die Darstellung ist jedoch assymetrisch. Die 0 hat eine eindeutige Darstellung $[0, \ldots, 0]$ und ist im Bereich der positiven Zahlen angesiedelt, welcher dadurch von 0 bis $2^{n-1}-1$ reicht, während die negativen Zahlen von -2^{n-1} bis -1 reichen. Für $-2^{n-1} = [1, 0, \ldots, 0]$ gibt es daher kein Komplement.

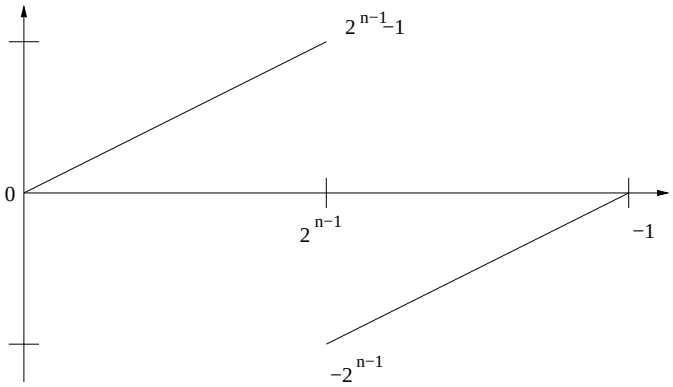

Abbildung 1: Wert der Zweier-Komplement-Zahlen

2.3 Binäre Entscheidungs Diagramme

Binäre Entscheidungsdiagramme gehen zurück auf eine Arbeit von Bryant [1] und sind eine Form der Repräsentation aussagenlogischer Formeln. Sie basieren darauf, dass der if-then-else Operator eine Basis bildet, d.h. sich jede aussagenlogische Formel ausschließlich durch if-then-else darstellen läßt.

$$\begin{aligned}\neg\varphi &\equiv (\varphi \Rightarrow 0 \mid 1)\\ \varphi \wedge \psi &\equiv (\varphi \Rightarrow \psi \mid 0)\\ \varphi \vee \psi &\equiv (\varphi \Rightarrow 1 \mid \psi)\end{aligned}$$

Ein Entscheidungsdiagramm ergibt sich indem in einer Formel jedes Auftreten eines Operators durch if-then-else ersetzt wird. Werden die in der Formel vorhandenen Variablen geordnet und gleiche Teilbäume nur einmal gespeichert, so bilden die BDDs eine kanonische Normalform.

Definition 8 (Geordnete BDDs). *Seien die Variablen $\mathcal{V} = \{x_1, x_2, \ldots, x_n\}$ mit einer bijektiven Funktion $\pi : \mathcal{V} \to \{2, \ldots, \mid \mathcal{V} \mid +1\}$ gegeben.*
Sei degree : $\mathcal{V} \cup \mathbb{N} \to \mathbb{N}$

$$\textit{degree}(v) := \begin{cases} \pi(v) & \textit{: falls } v \in \mathcal{V} \\ n & \textit{: falls } n \in \mathbb{N} \end{cases}$$

so gilt mit
$x \prec_\pi y :\Leftrightarrow \textit{degree}(x) < \textit{degree}(y)$
dass

$0 \prec_\pi 1 \prec_\pi x_1 \prec_\pi \ldots$

Ein geordnetes binäres Entscheidungsdiagramm (OBDD) über den Variablen $\mathcal{V}$ bezüglich der Ordnung $\prec_\pi$ ist ein Tupel G = ($\mathcal{V} \cup \{L_0, L_1\}$, v_0 , label, high, low) mit

$\forall\, v \in \mathcal{V}$.*label(*$v$*)* $\in \mathcal{V}$
- *label(*L_0*)* = 0 *und label(*L_1*)* = 1
- *high :* $\mathcal{V} \rightarrow \mathcal{V} \cup \{\, L_0, L_1 \,\}$ *und*
- *low :* $\mathcal{V} \rightarrow \mathcal{V} \cup \{\, L_0, L_1 \,\}$ *mit*
  - *degree(label(high(*v*)))* < *degree(label(*v*))*
  - *degree(label(low(*v*)))* < *degree(label(*v*))*
- $\exists\, v_0 \in \mathcal{V}.\ \forall\, v \in \mathcal{V}$.*high(*$v$*)* $\neq v_0 \wedge$ *low(*v*)* $\neq v_0$

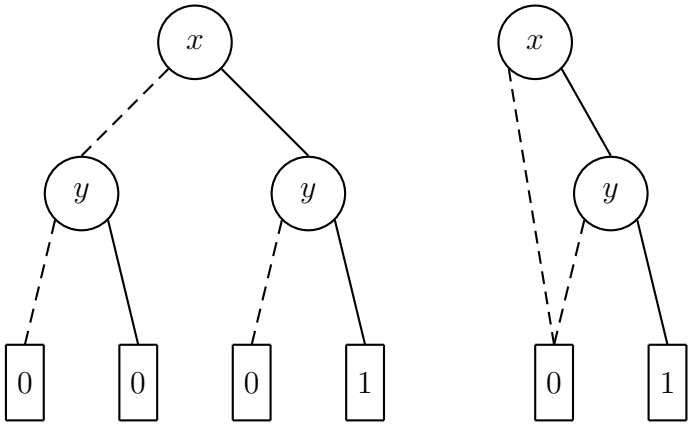

Abbildung 2: BDDs zu $x \wedge y$

Es gibt einen Wurzelknoten, v_0, von dem aus sich das Entscheidungsdiagramm aufbaut. Die Auswertung eines OBDD erfolgt indem man entsprechend der Belegung des Wurzelknotens der High bzw. Low Kante folgt, und dieses Vorgehen fortsetzt bis man an einem Blatt angekommen ist. Das Label des Blatts gibt dann die Gültigkeit des OBDDs für die Belegung an. Beide OBDDs aus Abbildung 1 beschreiben die Formel $x \wedge y$, wobei die High - Kante durch eine durchgezogene und die Low - Kante durch eine gestrichelte Linie dargestellt sind.
Nun müssen noch gleiche Teilbäume entfernt werden.

Definition 9 (Isomorphe OBDDs). *Zwei OBDDs $\mathcal{B}_1$ und $\mathcal{B}_2$ heißen isomorph, wenn es eine bijektive Abbildung $\Theta : \mathcal{V}_1 \rightarrow \mathcal{V}_2$ gibt, so dass für jeden Knoten gilt:*
falls v ein Blatt ist, so ist auch $\Theta(v)$ ein Blatt und trägt dieselbe Markierung,
falls v kein Blatt ist, so gilt
- label(v) = label($\Theta(v)$)
- (Θ(high(v)) = high($\Theta(v)$)
- (Θ(low(v)) = low($\Theta(v)$)

Definition 10 (Reduzierte OBDDs). *Ein OBDD ist reduziert (ein ROBDD), wenn es*
- keinen Knoten v gibt mit high(v) = low(v)
- kein Paar von Knoten v_1, v_2 gibt, so dass die in v_1 und v_2 wurzelnden Untergraphen isomorph sind.

In Abbildung 2 ist zu dem OBDD zu $x \wedge y$ die reduzierte Form angegeben. In der Praxis werden nur reduzierte OBDDs eingesetzt, da sie einen geringeren Speicherplatzbedarf aufweisen, und eine kanonische Normalform für die aussagenlogischen Formeln sind.
Eine weitere nützliche Eigenschaft betrifft die Überprüfung auf Gleichheit von Formeln. Da äquivalente Formeln nur einmal gespeichert werden, reduziert sich das Problem auf einen simplen Zeigervergleich. Des Weiteren lassen sich die booleschen Operationen effizient auf BDDs ausführen. Und es existieren Heuristiken für Variablenordnungen π, um die Größe der BDDs zu minimieren.

2.4 Presburger Arithmetik

Die Presburger Arithmetik entstammt der Arbeit „Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige hervortritt“ von Moj$\dot{z}$esz Presburger [2]. Presburger zeigte durch Quantorenelimination, dass die Arithmetik der Natürlichen Zahlen mit der Addition als einziger erlaubter Operation sowohl Vollständig als auch Entscheidbar ist.

Definition 11 (Syntax der Presburger Arithmetik). *Sei $\mathcal{V}$ eine endliche Menge an Variablen, dann ist die Menge der Presburger Terme $Term_{\mathcal{V}}$ wie folgt definiert:*
Sei τ, $\pi \in Term_{\mathcal{V}}$, dann ist auch
$\mathbb{Z} \in Term_{\mathcal{V}}$
$\mathcal{V} \in Term_{\mathcal{V}}$
$\tau + \pi$, $\tau - \pi \in Term_{\mathcal{V}}$

$c \cdot \tau \in Term_{\mathcal{V}}$, *falls* $c \in \mathbb{Z}$

Die Menge der Presburger Formeln $Pres_{\mathcal{V}}$ *ist definiert als:*
Sei $\tau, \pi \in Term_{\mathcal{V}}$, $\varphi, \psi \in Pres_{\mathcal{V}}$, *dann ist auch*
$\tau \leq \pi \in Pres_{\mathcal{V}}$
$\neg\, \varphi \in Pres_{\mathcal{V}}$
$\varphi \wedge \psi \in Pres_{\mathcal{V}}$
$\exists v.\varphi \in Pres_{\mathcal{V}}$, *falls* $v \in \mathcal{V}$

Die ursprüngliche Arbeit beschränkte sich auf die Menge der natürlichen Zahlen, doch läßt sie sich leicht auf die Menge der ganzen Zahlen erweitern. Die Multiplikation mit ganzen Zahlen stellt keinen Widerspruch zur Beschränkung auf die Addition als ausschließlicher Operation dar, da sie sich durch endlich viele Additionen darstellen läßt.

Definition 12 (Semantik der Presburger Arithmetik). *Sei* $\varsigma : \mathcal{V} \to \mathbb{Z}$ *eine Variablenbelegung für eine endliche Variablenmenge* $\mathcal{V}$. *Darauf basierend definieren wir die Erweiterung* $\varsigma^* : Term_{\mathcal{V}} \to \mathbb{Z}$
Sei $c \in \mathbb{Z}$, $\tau, \pi \in Term_{\mathcal{V}}$, $\varphi, \psi \in Pres_{\mathcal{V}}$
$\varsigma^*(c) := c$
$\varsigma^*(v) := \varsigma(v)$
$\varsigma^*(\tau + \pi) := \varsigma^*(\tau) + \varsigma^*(\pi)$
$\varsigma^*(\tau - \pi) := \varsigma^*(\tau) - \varsigma^*(\pi)$
$\varsigma^*(c \cdot \tau) := c \cdot \varsigma^*(\tau)$

Darauf aufbauend definieren wir die Erweiterung $\varsigma_{\varsigma^*} : Pres_{\mathcal{V}} \to \mathbb{B}$

$$
\begin{aligned}
\varsigma_{\varsigma^*}(\tau \leq \pi) &:= 1 \textit{ genau dann wenn } \varsigma^*(\tau) \leq \varsigma^*(\pi) \\
\varsigma_{\varsigma^*}(\neg\varphi) &:= \begin{cases} 1 \textit{ falls } \varsigma^*(\varphi) = 0 \\ 0 \textit{ falls } \varsigma^*(\varphi) = 1 \end{cases} \\
\varsigma_{\varsigma^*}(\varphi \wedge \psi) &:= \begin{cases} 1 \textit{ falls } \varsigma_{\varsigma^*}(\varphi) = 1 und \varsigma_{\varsigma^*}(\psi) = 1 \\ 0 \textit{ sonst} \end{cases} \\
\varsigma_{\varsigma^*}(\exists v.\varphi) &:= 1 \textit{ genau dann wenn es ein } c \in \mathbb{Z} \textit{ gibt so dass } \varsigma_{\varsigma[v/c]^*}(\varphi) = 1
\end{aligned}
$$

Selbstverständlich lassen sich durch Ausnutzung von Gesetzmäßigkeiten wie zum Beispiel den De Morganschen Gesetzen weitere Operatoren definieren. Einige sind:
$\varphi \vee \psi := \neg(\neg\varphi \wedge \neg\psi)$
$\tau = \pi := (\tau \leq \pi) \wedge (\pi \leq \tau)$
$\tau \neq \pi := \neg(\tau = \pi)$

2.4.1 Das Entscheidungsverfahren nach Cooper

Obwohl Presburger ein Entscheidungsverfahren lieferte dass durch Quantorenelimination entscheiden kann ob eine Presburger Formel in der keine freien Variablen vorkommen, d.h. jedes Vorkommen einer Variable durch einen Quantor gebunden ist, so ist das Verfahren ineffizient. Sollten freie Variablen vorkommen kann die Formel geschlossen werden in dem die freien Variablen allquantifiziert werden. 1972 wurde von D.C. Cooper ein Verfahren vorgestellt, welches deutlich effizienter arbeitet [3].
Dabei ist zu beachten dass einer Presburger Formel $\exists v.\varphi$ keine quantorenfreie Presburger Formel entspricht. Dieses Problem lässt sich umgehen indem Kongruenzrelationen $\equiv_m, m > 0$ eingeführt werden, die durch die Quantorenelimination auftreten.
Coopers Verfahren arbeitet auf einer Formel $\exists v.\varphi$, wobei φ quantorenfrei ist. Es besteht aus den folgenden Schritten:
1. Die Formel wird in Negationsnormalform gebracht und dabei die Negationen so weit wie möglich nach innen geschoben. Dann werden die Atome durch solche ersetzt in denen $<$, $\equiv_c$ und $\not\equiv_c$ vorkommen.
2. Die quantifizierte Variable wird separiert und die sie enthaltenden Atome so vereinfacht dass sie nur noch eine der folgenden vier Formen haben, wobei c,$m > 0$ und $x \notin Var(\tau)$
(1) $c \cdot x < \tau$
(2) $\tau < c \cdot x$
(3) $c \cdot x \equiv_m \tau$
(4) $c \cdot x \not\equiv_m \tau$
3. Sei d das kleinste gemeinsame Vielfache der Koeffizienten von x, dann werden alle Koeffizienten und Moduln so erweitert dass die Koeffizienten gleich d sind.
4. Sei n das kleinste gemeinsame Vielfache der Moduln und sei $\varphi_{-\infty}$ die Formel φ in der Atome des Typs (1) durch 1, und solche des Typs (2) durch 0 ersetzt wurden. Analog sei $\varphi_{-\infty}$ die Formel φ in der Atome des Typs (1) durch 0, und solche des Typs (2) durch 1 ersetzt wurden. Je nachdem welche Menge größer ist wird $\exists v.\varphi$ ersetzt durch:
$\bigvee_{i=1}^{n}([\varphi_\infty]_x^{-i} \vee \bigvee_{a\in A}[\varphi]_x^{a-i})$ falls die Menge A der Atome des Typs (1) größer ist, oder
$\bigvee_{i=1}^{n}([\varphi_{-\infty}]_x^{i} \vee \bigvee_{b\in B}[\varphi]_x^{b+i})$ falls die Menge B der Atome des Typs (2) größer ist.
1974 wurde von Fischer und Rabin gezeigt dass das Entscheidungsverfahren nach Cooper als untere Schranke doppelt exponentielle Laufzeit aufweist [4]. Im Jahre 1978 konnte Oppen nachweisen dass die obere Schranke bei dreifach exponenti-

eller Laufzeit liegt [5].

2.4.2 Automatenbasierte Entscheidungsverfahren

Ein vollkommen anderer Ansatz Presburger Formeln zu entscheiden ist die Übersetzung der Formel φ in einen Automaten der die Modelle der Formel kodiert. Automaten können für Presburger Formeln in der gleichen Weise als kanonische Repräsentationen benutzt werden wie BDDs für aussagenlogische Formeln benutzt werden. Dieser Ansatz geht zurück auf die Arbeit von Büchi 1962 [7], in der er zeigte dass durch die Übersetzung in Automaten monadische Theorien mit einem Nachfolger entschieden werden können. Er zeigte auch dass diese Übersetzung auch im besten Fall nicht elementare Komplexität aufweist.
Spätere Untersuchungen legen die Vermutung nahe dass die Automatenkonstruktion übermäßig pessimistisch bewertet wird [8]. Auch zeigt der Einsatz in der Praxis dass die Verwendung von Automaten durchaus praktikabel ist. Entsprechend werden Automaten für die Modellprüfung eingesetzt [10] und auch die Modellprüfung von μ-Kalkül Formeln auf Kripkestrukturen mit unendlich großen Datentypen profitiert von der Nutzung von Automaten [12] [13].
Die Konstruktion der Automaten basiert auf der Übersetzung von einzelnen Gleichungen und Ungleichungen und der anschließenden Verknüpfung der Automaten entsprechend der zu übersetzenden Formel.
Im ersten Schritt wird die Formel in die Form einer Summe gebracht $\sum_{i=1}^{n} a_i v_i = c$, wobei $v \in \mathcal{V}, n =\mid \mathcal{V} \mid$ und $a_i, c \in \mathbb{Z}$. Die Zustände des Automaten entsprechen Integern die wiederum der linken Seite der Gleichung mit den bis zu einem Zeitpunkt gelesenen Bits entsprechen. Der einzige akzeptierende Zustand entspricht der rechten Seite der Gleichung c. Die vom Automaten gelesenen Buchstaben sind Bitvektoren die zu jeder Variable ein Bit liefern. Die Variablenbelegungen werden durch Zweier-Komplement-Zahlen kodiert.
Gegeben sei der Automat $\mathcal{A} = (\mathcal{Q} \cup \{q^0\}, \mathbb{B}^n, \Delta, q^0, \{c\})$. Die Zustandsmenge der Integer wird dabei um einen Initialzustand q^0 erweitert der verhindert dass der Automat das leere Wort akzeptiert.
Sei $a.b = \sum_{i=1}^{n} a_i \cdot b_i$,dann ist die Transitionsrelation $\Delta : \mathcal{Q} \cup q^0 \times \mathbb{B}^n \rightarrow \mathcal{Q}$ definiert als:
$\Delta(q^0, b) = -a.b$
$\Delta(q, b) = 2q + a.b$, falls $q \neq q^0$
Der Beispiel-Automat in Abbildung 4 akzeptiert alle im Zweier-Komplement kodierten Lösungen der Gleichung $x - y = z$. Für den vorgestellten Algorithmus umgeformt lautet die Gleichung $x - y - z = 0$, bzw. $1 \cdot x + 1 \cdot y - 1 \cdot z = 0$. Damit

```
function Construct-Equation (A, c)
    Q := {q^0, c};
    L := {c};
    Δ := {};
    while (L ≠ ∅) do
        s := selectfrom(L);
        L := L \ s;
        for every b ∈ 2^n do
            s_0 = (s - a.b) / 2;
            if (s_0 ∈ ℤ) then
                if (s_0 ∉ Q) then
                    Q = Q ∪ {s_0};
                    L = L ∪ {s_0};
                end;
                Δ(s_0, b) := s;
            end;
            if (s = -a.b) then
                Δ(q^0, b) := s;
            end;
        end;
    end;
    Q := Q ∪ q^0;
    return (Q ∪ {q^0}, B^n, Δ ,q^0, {c});
end
```

Abbildung 3: Konstruktion eines Automaten für eine Gleichung

ergeben sich die Faktoren a_i zu [1,-1,-1] und $c = 0$. Die Tupel an den Transitionen des Automaten entsprechen den bisher gelesenen Buchstaben des Wortes. Die einzelnen Stellen der Tupel entsprechen dabei den Bits der Belegungen der Variablen x, y und z.
Es bleibt das Problem dass Buchstaben die zu einem nicht akzeptierten Wort gehören nicht im Algorithmus berücksichtigt werden. Um dieses Problem zu beseitigen wird ein weiterer Sonderzustand q^{-1} eingeführt der genau dann angenommen wird wenn das Wort nicht akzeptiert wird. Eine Vervollständigung des Automaten erweitert die Transitionsrelation Δ so dass in jedem Zustand der Buchstabe b der noch nicht in Δ ist zu dem nichtakzeptierenden Zustand q^{-1} führt. Der resultie-

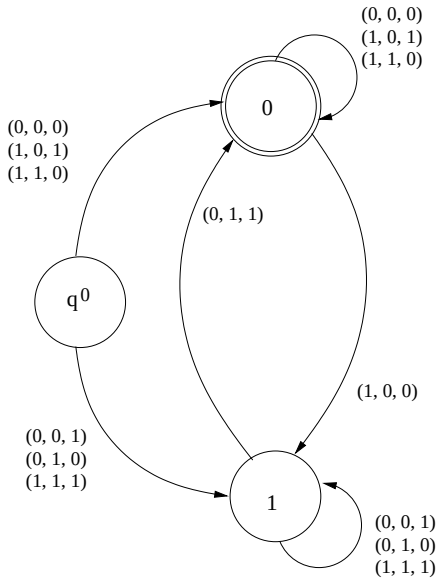

Abbildung 4: Automat für die Gleichung x - y = z

rende Automat ist in Abbildung 5 zu sehen
Die Konstruktion eines Automaten für Ungleichungen ist etwas komplizierter, wird jedoch vom gleichen Algorithmus durch zwei Anpassungen erledigt.
Die Menge der akzeptierenden Zustände ist im Falle einer Ungleichung entsprechend $\mathcal{C} = \{s \mid s \leq c\}$ da jede Lösung der Gleichung die kleiner oder gleich c ist eine gültige ist. Dies gilt natürlich auch für Transitionen $s_0 = \frac{s-a.b}{2}$ deren Ergebnis nicht in $\mathbb{Z}$ aber dennoch kleiner gleich c ist. Daher wird diese Transition zum Zustand $s_0 = \lfloor \frac{s-a.b}{2} \rfloor$ geleitet. Dies gilt natürlich auch für Transitionen vom Initialzustand, weshalb $-a.b \leq s$ eine Transition vom Initialzustand bedingt.
Diese Konstruktion hat den Nachteil dass der erzeugte Automat nichtdeteministisch ist, da es mehrere Zustände s gibt für die $\lfloor \frac{s-a.b}{2} \rfloor$ den gleichen Wert besitzt. Boigelot und Wolper konnten jedoch zeigen [8], dass diese Automaten eine bestimmte Struktur besitzen die es erlaubt sie in linearer Zeit zu determinisieren. Das Vorgehen dabei beruht darauf dass wenn ein Zustand s Transitionen zu den Zuständen s_1 und s_2 besitzt o.B.d.A. angenommen werden kann dass $s_1 < s_2$ gilt und zur Entscheidung ob das Wort akzeptiert wird nur die strengere Transition zu s_1 benötigt wird.
Nachdem die Automaten für Gleichungen und Ungleichungen konstruiert sind

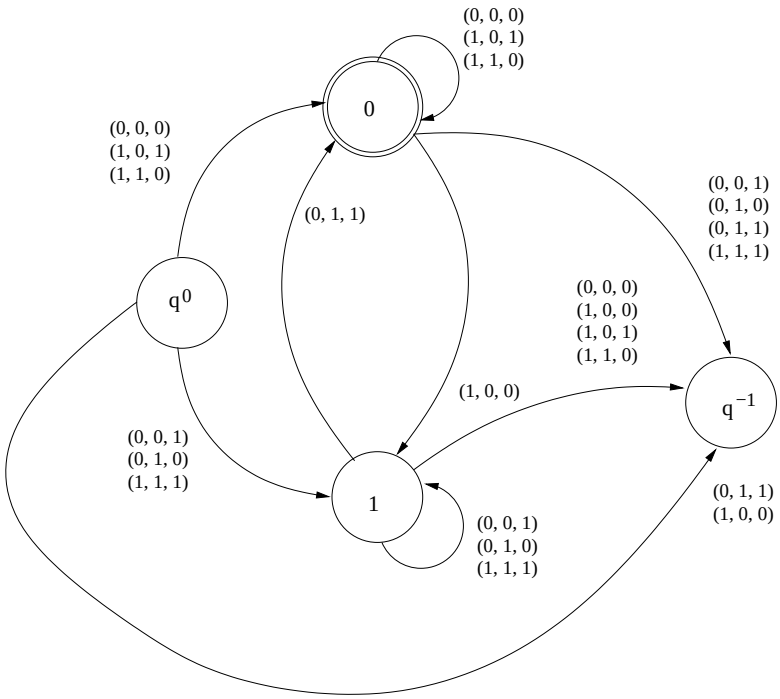

Abbildung 5: Vollständiger Automat für die Gleichung x - y = z

müssen noch die boolschen Verknüpfungen $\neg$ und $\wedge$ sowie die Quantifizierung definiert werden.
Sei $\mathcal{A}_\varphi = (\mathcal{Q}_\varphi, \mathbb{B}^n, \Delta_\varphi, \mathcal{I}_\varphi, \mathcal{C}_\varphi)$ und $\mathcal{A}_\psi = (\mathcal{Q}_\psi, \mathbb{B}^n, \Delta_\psi, cI_\psi, \mathcal{C}_\psi)$, dann ist
$\neg\mathcal{A}_\varphi := (\mathcal{Q}_\varphi, \mathbb{B}^n, \Delta_\varphi, q^0_\varphi, \mathcal{Q}_\varphi \setminus \mathcal{C}_\varphi)$
$\mathcal{A}_\varphi \wedge \mathcal{A}_\psi := (\mathcal{Q}_\varphi \times \mathcal{Q}_\psi, \mathbb{B}^n, \Delta, \mathcal{I}_\varphi \times \mathcal{I}_\psi, \mathcal{C}_\varphi \times \mathcal{C}_\psi)$ wobei $((s_\varphi, s_\psi), b, (t_\varphi, t_\psi)) \in \Delta$ genau dann wenn $(s_\varphi, b, t_\varphi) \in \Delta_\varphi$ und $(s_\psi, b, t_\psi) \in \Delta_\psi$
Die Quantifizierung beruht auf der Ersetzung jedes Vorkommens von $\forall\varphi$ durch $\neg\exists\neg\varphi$. Anschließend läßt sich durch Anwendung der Projektion das Auftreten der quantifizierten Variable beseitigen.
Für die Projektion gilt, dass (s, (b_n, ..., b_{i+1}, b_i, b_{i+1}, ..., b_0), t) der Transitionsrelation nach der Projektion ist wenn b_i die projizierte Variable ist und es ein b gibt so dass (s, (b_n, ..., b_{i+1}, b, b_{i+1}, ..., b_0), t) die Transitionsrelation vor der Projektion ist. Das Vorgehen ist in [8] an einem einfachen Beispiel leicht nachzuvollziehen, welches in Abbildung 7 zu sehen.
Der obere Automat akzeptiert das Wort für $x = 1 \wedge y = 4$, im Unteren wurde

```
function Construct-Inequation (A, c)
    Q := {q^0, c};
    L := {c};
    Δ := {};
    C := {c};
    while (L ≠ ∅) do
        s := selectfrom(L);
        L := L \ s;
        for every b ∈ 2^n do
            s_0 = ⌊(s - a.b) / 2⌋;
            if (s_0 ∈ ℤ) then
                if (s_0 ∉ Q) then
                    Q = Q ∪ {s_0};
                    L = L ∪ {s_0};
                end;
                Δ(s_0, b) := s;
            end;
            if (-a.b ≤ s) then
                Δ(q^0, b) := s;
            end;
            if (s ≤ c) then
                C = C ∪ s;
            end;
        end;
    end;
    Q := Q ∪ q^0;
    return (Q ∪ {q^0}, 𝔹^n, Δ ,q^0, C);
end
```

Abbildung 6: Konstruktion eines Automaten für eine Ungleichung

die Variable y projiziert. Er akzeptiert noch das Wort $x = 1$, und die Belegung von y ist unerheblich. Die Variable y wurde aus dem Automaten herausquantifiziert. Dieses Vorgehen führt aber in der Regel zu nicht deterministischen Automaten für die die Determinisierung eine Komplexität von 2^n aufweist. Genauere Betrachtungen zeigen dass mehrfach hintereinander ausgeführte Projektionen keine deterministischen Automaten benötigen. Erst die Negation welche für den

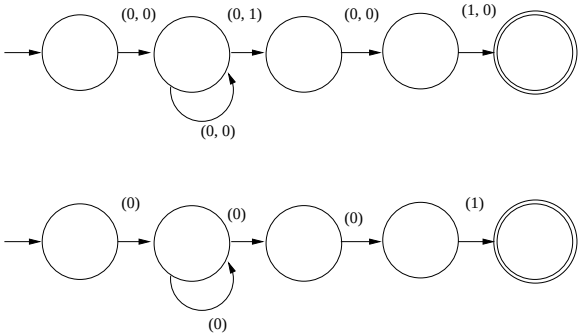

Abbildung 7: Automat für x=1 und y=4 und dessen Projektion

Allquantor benötigt wird braucht den Determinisierungsschritt, weshalb für jede Quantorenalternation ein exponentieller Bearbeitungsschritt nötig wird.
Das Beispiel zeigt jedoch auch dass durch die Projektion ein Problem auftreten kann. Der projezierte Automat akzeptiert zwar Wörter für $x = 1$ jedoch nur solche deren Länge mindestens 4 beträgt. Nach der Projektion muss der Automat so geändert werden, dass wenn ein Wort b^+w akzeptiert wird, auch bw akzeptiert wird. Dazu müssen die Menge der Zustände die durch b vom initialen Zustand aus erreichbar sind um jene Zustände erweitert werden die durch b^+ erreichbar sind.

3 SD-Automaten und Presburger Arithmetik

In diesem Kapitel werden nun die von Tobias Schüle vorgestellten Signed-Digit-Automaten [11] näher auf ihre Eignung zur Entscheidung von Presburger Arithmetik untersucht. Dazu werden zunächst die auf den Eigenschaften der SD-Zahlen basierenden Automaten und anschließend die für die Implementierung nötigen Schritte beschrieben.

3.1 SD-Automaten

Ein Signed Digit Automat $\mathcal{A}$ unterscheidet sich von den gewöhnlichen Automaten dadurch dass er grundsätzlich die Form eines Fächers aufweist. Es gibt keine Transitionen von einem Zustand q_1 zu einem Zustand q_2, sondern nur vom Initialzustand q^0 zu den einzelnen Zuständen und solche die von einem Zustand q wieder zu diesem zurückführen.

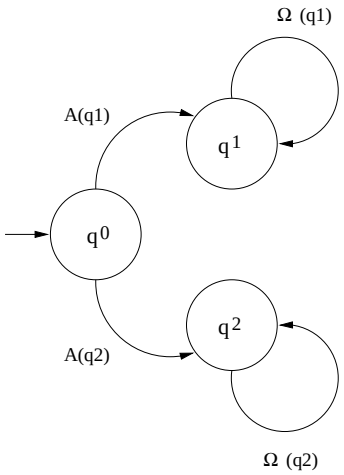

Abbildung 8: Beispiel für einen SD-Automaten

Definition 13 (Signed-Digit-Automat). *Ein Signed-Digit-Automat ist ein Tupel* $\mathcal{A} = (\mathcal{Q}, \mathcal{D}^{2n}, A, \Omega)$ *mit*
$\mathcal{Q}$ *einer Menge von Zuständen*
$\mathcal{D}^{2n}$ *einem Alphabet*
$A : \mathcal{Q} \rightarrow \mathcal{P}(\mathcal{D}^{2n})$ *einer Funktion die jedem Zustand* $q \in \mathcal{Q}$ *eine Menge initialer*

Transitionen zuordnet.
$\Omega : \mathcal{Q} \to \mathcal{P}(\mathcal{D}^{2n})$ einer Funktion die jedem Zustand $q \in \mathcal{Q}$ eine Menge finaler Transitionen zuordnet.

Der Initialzustand q^0 muss dabei nicht mit modeliert werden.

3.1.1 Wörter, Konsistenz und Akzeptanz

Die für die Addition nützliche Eigenschaft dass Summen- und Übertragsziffern an der Stelle i nur von den Ziffern an der Stelle i und $i-1$ abhängig sind bedingt dass Signed-Digit-Automaten andere Wörter lesen als die bisher verwendeten Automaten.
Es gibt n Variablen und aus der Definition kann man ersehen dass die gelesenen Buchstaben die doppelte Breite haben.

Definition 14 (Konsistenzeigenschaft). *Sei $l \in \mathcal{D}^{2n}$ ein Buchstabe. Sei $\lambda_i(l)$ das i-te Element des Buchstaben l. Sei $w = [l_m, \ldots, l_0]$ ein Wort. Dann gilt:*
Das Wort w ist Konsistent genau dann wenn w das Prädikat $con(w)$ erfüllt, mit $con(w) := \forall i \in \mathbb{N}.0 \leq i < n \to \lambda_{n+i}(l_0) = 0 \wedge \forall j \in \mathbb{N}.0 \leq j < m \to \lambda_{n+i}(l_j) = \lambda_{n+i}(l_{j-1})$

Diese Art des Lesens des Wortes sorgt dafür dass beim Lesen des i-ten Buchstabens auch die Belegungen der Variablen aus dem $i-1$ten Buchstabens bekannt sind. Der offensichtliche Nutzen liegt darin dass mit jedem gelesenen Buchstaben auch die dazugehörige Summe und der Übertrag gebildet werden können.
Für ein kurzes Beispiel sei $n = 2$ und die Basis $B = 3$. Das konsistente Wort welches die Variablen x mit 14 und y mit 37 belegt ist:

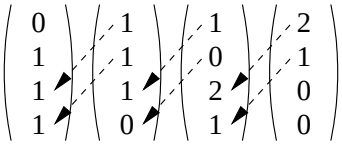

Abbildung 9: Beispiel für ein konsistentes Wort

Es ist nun die Frage welche Wörter w von einem Automaten $\mathcal{A}$ akzeptiert werden.

Definition 15 (Akzeptanz eines Zustandes). *Sei $\mathcal{A}$ ein Automat mit $\mathcal{A} = (\mathcal{Q}, \mathcal{D}^{2n}, A, \Omega)$, und w ein Wort mit $w = [l_m, \ldots, l_0]$. Dann akzeptiert der*

Zustand q das Wort w, wenn das Prädikat $acc_{\mathcal{A}}(q,w) \Leftrightarrow l_m \in A(q) \wedge \forall i \in \mathbb{N}.0 \leq i < m \rightarrow l_i \in \Omega(q)$ durch q und w erfüllt wird.

Ein einzelner Zustand akzeptiert also ein Wort wenn der höchstwertige Buchstabe in der initialen Transition, und alle weiteren Buchstaben in der finalen Transition enthalten sind. Die Erweiterung auf die Akzeptanz eines Automaten fällt entsprechend leicht, indem ein Automat ein Wort w genau dann akzeptiert, wenn er einen Zustand enthält der w akzeptiert.

Definition 16 (Akzeptanz eines Automaten). *Sei $\mathcal{A}$ ein Automat mit $\mathcal{A} = (\mathcal{Q}, \mathcal{D}^{2n}, A, \Omega)$, und w ein Wort mit $w = [l_m, \ldots, l_0]$. Dann akzeptiert der Automat $\mathcal{A}$ das Wort w, wenn es ein $q \in \mathcal{Q}$ gibt, der das Prädikat $acc_{\mathcal{A}}(q,w)$ erfüllt. Es gilt:*
$acc_{\mathcal{A}}(w) := \exists q \in \mathcal{Q}.acc_{\mathcal{A}}(q,w)$

Die Sprache des Automaten besteht aus der Menge von Wörtern die vom Automaten akzeptiert wird und konsistent ist.

Definition 17 (Sprache). *Die Sprache eines Automaten $\mathcal{A}$ ist $\mathcal{L} := \{w \in (\mathcal{D}^{2n})^* \mid con(w) \wedge acc_{\mathcal{A}}(w)\}$*

3.1.2 Automatenoperationen

Auf SD-Automaten sind die Verknüpfungen der Konjunktion und Disjunktion wie folgt definiert:

Definition 18 (Konjunktion). *Die Konjunktion zweier Automaten $\mathcal{A}_1 = (\mathcal{Q}_1, \mathcal{D}^{2n}, A_1, \Omega_1)$ und $\mathcal{A}_2 = (\mathcal{Q}_2, \mathcal{D}^{2n}, A_2, \Omega_2)$ ist definiert als $\mathcal{A}_1 \wedge \mathcal{A}_2 := (\mathcal{Q}_1 \times \mathcal{Q}_2, \mathcal{D}^{2n}, A_{12}, \Omega_{12})$ mit $A_{12}(q_1,q_2) = A_1(q_1) \cap A_2(q_2)$ und $\Omega_{12}(q_1,q_2) = \Omega_1(q_1) \cap \Omega_2(q_2)$*

Definition 19 (Disjunktion). *Die Disjunktion zweier Automaten $\mathcal{A}_1 = (\mathcal{Q}_1, \mathcal{D}^{2n}, A_1, \Omega_1)$ und $\mathcal{A}_2 = (\mathcal{Q}_2, \mathcal{D}^{2n}, A_2, \Omega_2)$ ist definiert als $\mathcal{A}_1 \vee \mathcal{A}_2 := (\mathcal{Q}_1 \cup \mathcal{Q}_2, \mathcal{D}^{2n}, A_1 \cup A_2, \Omega_1 \cup \Omega_2)$*

Die Disjunktion ist eine sehr einfache Operation da die Zustands- und Transitionsmengen der beiden verknüpften Automaten nur zusammengefasst werden müssen. Der erzeugte Automat akzeptiert dann die Wörter die $\mathcal{A}_1$ akzeptiert ebenso wie die, die von $\mathcal{A}_2$ akzeptiert werden.

Der Automat der Konjunktion akzeptiert nur was von beiden Automaten $\mathcal{A}_1$ und $\mathcal{A}_2$ akzeptiert wird. Daher müssen alle Transitionen der beiden Automaten miteinander geschnitten werden.
Das Quantifizieren eines SD-Automaten ist ebenso wie dessen Negation leider nicht möglich. Da für die Automatenkonstruktion der Existenzquantor benötigt wird werden quantifizierte Variablen wie freie Variablen behandelt.

3.1.3 Automatenkonstruktion

Die Automatenkonstruktion beruht auf der Erzeugung von Automaten für bestimmte Prädikate, die verknüpft werden können um die von den gewohnten Automatenklassen benutzten Gleichungen und Ungleichungen zu repräsentieren.

Die Prädikate für die Addition, die Subtraktion und die Gleichheit von Variablen beruhen dabei auf der Eigenschaft der Signed Digit Zahlen, dass die Summen- und Übertragsziffern der i-ten Stelle nur von den i-ten und $i-1$ten Stellen abhängen. Durch die Konsistenzeigenschaft des gelesenen Wortes sind die $i-1$ten Stellen auch bekannt und die Summenziffer läßt sich wie folgt beschreiben:
$s(x, y, x', y') := x + y + t(x', y') - t(x, y) \cdot \mathcal{B}$ mit

$$t(x,y) := \begin{cases} +1 & : \text{falls } x + y \geq \mathcal{A} \\ -1 & : \text{falls } x + y \leq -\mathcal{A} \\ 0 & : \text{sonst} \end{cases}$$

Für die Addition zweier Variablen x und y und dem Ergebnis z ergibt sich also dass $s(x_i, y_i, x'_i, y'_i) = z_i$ ist, wenn x' die Variable ist die die Belegung von x an der Stelle $i-1$ angibt. Dies entspricht der geforderten Konsistenzeigenschaft der vom Automaten gelesenen Wörter. Um dies genauer zu beschreiben sei $\sigma : \mathcal{V} \to \{0, \cdots, n-1\}$ eine Funktion die zu einer Variable den Index im gelesenen Wort angibt. Dann ist $\sigma(x') = \sigma(x) + n$.

Definition 20 (Addition ADDC). *Das Prädikat $ADDC(x, y, z)$ wird übersetzt in den Automaten $\mathcal{A} = (\{q\}, \mathcal{D}^{2n}, A, \Omega)$ mit $A(q) = \mathcal{S} \cap \mathcal{I}$ und $\Omega(q) = \mathcal{S}$, wobei $\mathcal{S} = \{(d_0, \ldots, d_{2n-1}) \mid s(d_{\sigma(x)}, d_{\sigma(y)}, d_{\sigma(x)+n}, d_{\sigma(y)+n}) = d_{\sigma(z)}\}$ und $\mathcal{I} = \{(d_0, \ldots, d_{2n-1}) \mid t(d_{\sigma(x)}, d_{\sigma(y)}) = 0\}$.*

Der so konstruierte Automat akzeptiert jedoch nicht alle Lösungen der Gleichung $x + y = z$, sondern nur solche z deren Ziffern denen entsprechen die durch $s(d_{\sigma(x)}, d_{\sigma(y)}, d_{\sigma(x)+n}, d_{\sigma(y)+n})$ bestimmt sind. Dieses Problem läßt sich umgehen indem man überprüft ob es ein u gibt so dass $u = x + y$ und $u = z$ ist. Es wird

also ein Prädikat benötigt welches zwei Variablen auf Gleichheit prüft. Nun sind zwei Signed Digit Zahlen u und z genau dann gleich wenn $u - z = 0$ gilt.

Definition 21 (Gleichheit EQ). *Das Prädikat $EQ(x, y)$ wird übersetzt in den Automaten $\mathcal{A} = (\{q\}, \mathcal{D}^{2n}, A, \Omega)$ mit $A(q) = \mathcal{S} \cap \mathcal{I}$ und $\Omega(q) = \mathcal{S}$, wobei $\mathcal{S} = \{(d_0, \ldots, d_{2n-1}) \mid s(d_{\sigma(x)}, -d_{\sigma(y)}, d_{\sigma(x)+n}, -d_{\sigma(y)+n}) = 0\}$ und $\mathcal{I} = \{(d_0, \ldots, d_{2n-1}) \mid t(d_{\sigma(x)}, d_{\sigma(y)}) = 0\}$.*

Die Gleichheit mit Null läßt sich durch die einzigartige Darstellung der Null leichter überprüfen. Es muss nur jede Stelle gleich Null sein.

Definition 22 (Gleichheit EQ0). *$EQ0(x)$ wird übersetzt in den Automaten $\mathcal{A} = (\{q\}, \mathcal{D}^{2n}, A, \Omega)$ mit $A(q) = \Omega(q) = \{(d_0, \ldots, d_{2n-1}) \mid d_{\sigma(x)} = 0\}$.*

Mit den Prädikaten $ADDC$ und EQ kann nun also die Addition so übersetzt werden, dass sie korrekt alle Lösungen der Gleichung $x + y = z$ akzeptiert.

Definition 23 (Addition ADD). *Das Prädikat ADD setzt sich wie folgt zusammen: $ADD(x, y, z) := \exists u.ADDC(x, y, u) \wedge EQ(u, z)$.*

Die entsprechenden Prädikate für die Subtraktion werden analog definiert und basieren wie es schon bei der Gleichheit EQ benutzt wurde darauf dass das Komplement einer SD-Zahl $x = [x_m, \ldots, x_0]$ durch Vorzeichenumkehrung der einzelnen Ziffern gefunden wird: $-x = [-x_m, \ldots, -x_0]$.

Definition 24 (Subtraktion SUBC). *Das Prädikat $SUBC(x, y, z)$ wird übersetzt in den Automaten $\mathcal{A} = (\{q\}, \mathcal{D}^{2n}, A, \Omega)$ mit $A(q) = \mathcal{S} \cap \mathcal{I}$ und $\Omega(q) = \mathcal{S}$, wobei $\mathcal{S} = \{(d_0, \ldots, d_{2n-1}) \mid s(d_{\sigma(x)}, -d_{\sigma(y)}, d_{\sigma(x)+n}, -d_{\sigma(y)+n}) = d_{\sigma(z)}\}$ und $\mathcal{I} = \{(d_0, \ldots, d_{2n-1}) \mid t(d_{\sigma(x)}, d_{\sigma(y)}) = 0\}$.*

Definition 25 (Subtraktion SUB). *Das Prädikat SUB setzt sich wie folgt zusammen: $SUB(x, y, z) := \exists u.SUBC(x, y, u) \wedge EQ(u, z)$.*

Für die Übersetzung von Gleichungen und Ungleichungen fehlen nun noch Prädikate für $x < y$ und Monome $c \cdot x = y$. Für beide werden wiederum existenzielle Quantoren benötigt.

Definition 26 (Kleiner Null LT0C). *$LT0C(x)$ wird übersetzt in den Automaten $\mathcal{A} = (\{q\}, \mathcal{D}^{2n}, A, \Omega)$ mit $A(q) = \{(d_0, \ldots, d_{2n-1}) \mid d_{\sigma(x)} < 0\}$ und $\Omega(q) = \mathcal{D}^{2n}$.*

Der so erzeugte Automat akzeptiert nicht alle x die kleiner 0 sind, sondern lediglich jene die nicht mit einer führenden 0 beginnen. Diese Einschränkung wird jedoch durch die quantifizierte Verwendung in den Prädikaten Kleiner 0 $LT0$ und Kleiner LT aufgehoben.

Definition 27 (Kleiner Null LT0). $LT0(x) := \exists u.LT0C(u) \wedge EQ(u, x)$.

Definition 28 (Kleiner LT). $LT(x, y) := \exists u.SUB(x, y, u) \wedge LT0C(u)$.

Auch die Übersetzung von Monomen $c \cdot x = y$ in Signed-Digit-Automaten ist möglich. Das Vorgehen ist rekursiv definiert. Dabei wird c immer weiter verkleinert und die entsprechenden Vielfache von x aufaddiert.

Definition 29 (Monome MON).

$$MON(c, x, y) := \begin{cases} EQ0(y) & \text{: falls } c = 0 \\ \exists t.MON(\frac{c}{2}, x, t) \wedge ADD(t, t, y) & \text{: falls } c \text{ gerade} \\ \exists t.MON(\frac{c-1}{2}, x, t) \wedge \exists t'.ADD(t, t, t') \wedge & \\ \quad ADD(t', x, y) & \text{: sonst} \end{cases}$$

So lassen sich jedoch nur positive Koeffizienten übersetzen. Mit einer kleinen Erweiterung sind jedoch auch negative Koeffizienten möglich. Sie beruht darauf dass das Monom $-1 \cdot c \cdot x = y$ auch durch $c \cdot x = -1 \cdot y$ dargestellt werden kann. Es existiert folglich ein t sodass $c \cdot x = t$ und $t = -1 \cdot y$. Die Gleichheit von t und $-1 \cdot y$ wird überprüft dadurch dass die Summe von beiden 0 ergibt.

Definition 30 (Monome MON (erweitert)).

$$MON(c, x, y) := \begin{cases} EQ0(y) & \text{: falls } c = 0 \\ \exists t.MON(-1 \cdot c, x, t) \wedge \exists u.ADD(t, y, u) & \\ \quad \wedge EQ0(u) & \text{: falls } c < 0 \\ \exists t.MON(\frac{c}{2}, x, t) \wedge ADD(t, t, y) & \text{: falls } c \text{ gerade} \\ \exists t.MON(\frac{c-1}{2}, x, t) \wedge \exists t'.ADD(t, t, t') \wedge & \\ \quad ADD(t', x, y) & \text{: sonst} \end{cases}$$

Es ist bemerkenswert dass die zu den elementaren Prädikaten $ADDC$, $SUBC$, $EQ0$, EQ, $LT0C$ konstruierten Automaten aus nur einem Zustand bestehen. Die darauf aufbauenden ADD, SUB, $LT0$, LT und MON enthalten als einzige Verknüpfung der Automaten die Konjunktion, welche die Zustandsmenge nicht vergrößert.
Weiter ist zu bemerken dass es während der Konstruktion nicht möglich ist Konstanten zu verwenden. Ein Automat der $x = 3$ repräsentiert ist somit nicht direkt konstruierbar. Es muss für jede Konstante eine neue Variable eingeführt werden, und im Automaten muss eine Liste geführt werden die jeder dieser Variablen ihre Belegung zuordnet. Wird später der Automat auf Leerheit, oder Akzeptanz eines

Wortes überprüft muss dabei den Variablen in dieser Liste ihre Belegung zugewiesen werden.
Die vorgestellten Prädikate genügen nun um Gleichungen und Ungleichungen der folgenden Arten zu übersetzen:

$$\begin{array}{ll} 1) & \sum_{i=0}^{n} c_i \cdot x_i = c \\ 2) & \sum_{i=0}^{n} c_i \cdot x_i < c \\ 3) & \sum_{i=0}^{n} c_i \cdot x_i \neq c \end{array}$$

Dabei sind die ersten Schritte für alle drei Typen identisch, erst im letzten Schritt ergeben sich Unterschiede. Zunächst werden n neue freie Variablen eingeführt und entsprechend viele Automaten für die Monome erzeugt. Dann gibt es für $c_i \cdot x_i = y_i$ einen Automaten $MON(c_i, x_i, y_i)$. Nun müssen die neu eingeführten Variablen y_i addiert werden $\sum_{i=0}^{n-1} y_i = z$. Die linke Seite der zu übersetzenden Gleichung entspricht also der Variable z. Für die Konstante c wird eine weitere Variable u benutzt und ihr der Wert c zugewiesen. Erst hier im letzten Schritt zeigt sich der Unterschied zwischen den Gleichungstypen. Für Gleichungen (1) wird ein Automat $EQ(z, u)$ konstruiert, für Typ (2) wird entsprechend das Prädikat $LT(z, u)$ benutzt. Alle in diesem Prozess erzeugten Automaten werden durch Konjunktionen verknüpft, wodurch sich der Automat für die Gleichung ergibt. Diese Automaten für die Typen (1) und (2) bestehen folglich auch nur aus einem einzigen Zustand. Für den Typ (3) gilt dies nicht mehr da sich Signed Digit Automaten nicht negieren lassen. Um $z \neq u$ darstellen zu können muss man auf $z < u \vee u < z$ zurückgreifen. Die Disjunktion führt jedoch einen zusätzlichen Zustand ein. Das Vorgehen ist in Abbildung 10 zu sehen.
Ein Beispiel kann gegeben werden mit der Übersetzung der $4 \cdot x_0 + 4 \cdot x_1 + 4 \cdot x_2 = 3$. Der Algorithmus wird für jedes in der Gleichung vorkommende Monom eine neue, freie Variable einführen, diese seien y_0, y_1 und y_2. Mit Hilfe dieser Variablen werden die Prädikate für diese Monome konstruiert.

$$\begin{aligned} \mathcal{A}_1 &:= MON(4, x_0, y_0) \\ \mathcal{A}_2 &:= MON(4, x_1, y_1) \\ \mathcal{A}_3 &:= MON(4, x_2, y_2) \end{aligned}$$

Anschließend wird die Summe der Monome erzeugt. Dazu werden wieder zwei neue Variablen benötigt: z_1, z_2

$$\begin{aligned} \mathcal{B}_1 &:= ADD(y_0, y_1, z_1) \\ \mathcal{B}_2 &:= ADD(y_2, z_1, z_2) \end{aligned}$$

```
function Construct (C, X, rel, c)
    n = |X|;
    for (i=0; i<n; i++) do
        y_i = new-var;
        A_i = MON(c_i, x_i, y_i);
    end;
    if (n==1) do
        z = new-var;
        B_1 = EQ(z,y_1);
    else
        z_1 = new-var;
        B_1 = ADD(y_0,y_1,z_1);
        if (n>1) do
            for (i = 3; i ≤ n; i++) do
                z_{i-1} = new-var;
                B_{i-1} = ADD(y_{i-1},z_{i-2},z_{i-1});
            end;
        end;
        z=z_{n-1};
    end;
    u = new-var;
    assign-const(u, c);
    if (rel==eq) do
        R = EQ(z, u);
    else if (rel==le) do
        R = LE(z, u);
    else if (rel==eq) do
        R_1 = LE(z, u);
        R_2 = LE(u, z);
        R = R_1 ∨ R_2;
    end;
    A = (∧_{i=0}^{n-1} A_i) ∧ (∧_{i=1}^{n-1} B_i) ∧ R;
    return(A);
end
```

Abbildung 10: Konstruktion eines SD-Automaten

Die höchste erzeugte Variable z_2 enthält die Summe der Monome, und entspricht damit der linken Seite der Gleichung. Da die Konstante 3 nicht direkt dargestellt werden kann muss eine Variable u erzeugt werden, der der Wert 3 zugewiesen wird. Nun wird entsprechend der in der Gleichung verwendeten Relation das Prädikat erzeugt welches die linke Seite der Gleichung mit der rechten Seite verbindet.

$$\mathcal{R} := EQ(z_2, u)$$

Der Automat, der der gesamten Gleichung entspricht wird nun im letzten Schritt dadurch erzeugt dass alle während des Verfahrens konstruierten Automaten mit Konjunktionen verbunden werden.

$$\mathcal{A} := \mathcal{A}_1 \wedge \mathcal{A}_2 \wedge \mathcal{A}_3 \wedge \mathcal{B}_1 \wedge \mathcal{B}_2 \wedge \mathcal{R}$$

3.1.4 Akzeptanz und Leerheit

Um zu prüfen ob ein Automat $\mathcal{A} = (\{q\}, \mathcal{D}^{2n}, A, \Omega)$ ein gegebenes Wort $w = l_m \dots l_0$ akzeptiert muss das Prädikat $acc_{\mathcal{A}}(w)$ geprüft werden. Es muss also einen Zustand q geben, so dass für den ersten Buchstaben $l_m \in A(q)$ gilt und alle weitern $l_i \in \Omega(q)$ für $0 < i < m$.
Besondere Aufmerksamkeit muss dabei den Variablen die eine Konstante repräsentieren zukommen. Für jede Variable die als Platzhalter für eine Konstante dient muss das Wort w um die entsprechende Konstante erweitert werden. Falls die Konstanten durch Integer gegeben sind müssen sie natürlich zunächst in eine Signed-Digit Darstellung der entsprechenden Basis gebracht werden.
Das Wort muss dabei auch mit 0 beginnen und auch mit 0 in der Konsistenzeigenschaft enden, um zu gewährleisten dass keine Buchstaben vor l_m oder nach l_0 kommen.
Sei als Beispiel ein Automat zu $x = 5$ mit $B = 3$ gegeben. Wird nun geprüft ob $x = 3$ akzeptiert wird, so ist das fertige Wort einschließlich der Konsistenzeigenschaft gegeben durch:

$$\begin{matrix} x \\ c \\ x' \\ c' \end{matrix} \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \\ 2 \end{pmatrix} \begin{pmatrix} 2 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Sind im Automaten durch Quantifizierung aufgetretene freie Variablen vorhanden so müssen diese bei der Akzeptanzprüfung natürlich auch berücksichtigt werden. Da für sie in den Buchstaben keine Belegungen gegeben sind müssen diese für sie geraten werden. Dabei müssen die geratenen Belegungen die Konsistenzeigenschaft erfüllen und wie die in den Buchstaben gegebenen Belegungen in l_m und l_0 mit 0 beginnen und 0 in der Konsistenz enden.
Die Prüfung ob die Buchstaben in A und Ω enthalten sind läßt sich mit dem raten der Belegungen und der Prüfung der Konsistenz der freien Variablen zusammen durchführen in dem neue Variablen eingeführt werden und diese auf geeignete Weise mit den Variablen der A und Ω Mengen getauscht werden. Ist die Schnittmenge der durch das Tauschen entstandenen A und Ω Mengen nicht leer, so enthält sie die geratenen Belegungen der freien Variablen für dieses Wort. Diese Überprüfung muss für jeden Zustand q durchgeführt werden bis entweder alle geprüft wurden, oder aber einer gefunden wurde der das Wort akzeptiert.
Sei als Beispiel ein Automat gegeben dessen Variablen x und y sind, und der zusätzlich die freie Variable z besitzt. Sei weiterhin y ein Platzhalter für die Konstante $c = 5$. Wird nun das Wort für $x = 3$ geprüft so ist das Wort gegeben wie im vorherigen Beispiel.
Für die Variablenvertauschung werden dazu in diesem Beispiel 12 neue Variablen $e_0 \dots e_{11}$ benötigt. Die Schnittmenge ergibt sich dann zu:

$$\begin{array}{c} A(q)[x \setminus e_0, y \setminus e_1, z \setminus e_2, x' \setminus e_3, y' \setminus e_4, z' \setminus e_5] \cap \\ \Omega(q)[x \setminus e_3, y \setminus e_4, z \setminus e_5, x' \setminus e_6, y' \setminus e_7, z' \setminus e_8] \cap \\ \Omega(q)[x \setminus e_6, y \setminus e_7, z \setminus e_8, x' \setminus e_9, y' \setminus e_{10}, z' \setminus e_{11}] \end{array}$$

Die Variablen $e_0 \dots e_{11}$ müssen dann mit den durch das Wort gegebenen Werten belegt werden.

$$\begin{array}{cc} e_0 & \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ ? \end{pmatrix} \\ e_1 & \\ e_2 & \\ e_3 & \\ e_4 & \\ e_5 & \end{array} \quad \begin{array}{cc} e_3 & \begin{pmatrix} 1 \\ 1 \\ ? \\ 2 \\ 0 \\ ? \end{pmatrix} \\ e_4 & \\ e_5 & \\ e_6 & \\ e_7 & \\ e_8 & \end{array} \quad \begin{array}{cc} e_6 & \begin{pmatrix} 2 \\ 0 \\ ? \\ 0 \\ 0 \\ 0 \end{pmatrix} \\ e_7 & \\ e_8 & \\ e_9 & \\ e_{10} & \\ e_{11} & \end{array}$$

Ein Fragezeichen in dieser Variablenbelegung bezeichnet eine freie Variable, deren Belegung geraten werden muss. Die Mehrfachverwendung von Variablen, garantiert durch die Art der Überschneidung dass die Konsistenzeigenschaft des

```
function Accept (A, w)
    m = |w|;
    n = |Variables|;
    accepted = false;
    forall (q ∈ Q) do
        for (i=m; i>=0; i–) do
            l = w_i;
            Schnitt = ∅;
            if (n==m) do
                Schnitt = A(x_0\e_0 ... x_n\e_n);
                for (j=0; j<n; j++) do
                    if (x_j ≠ free) e_j = l_σ(x_j);
                end;
            else
                k = m - i;
                tmp = Ω(x_0\e_{n/2·k} ... x_n\e_{n+n/2·k});
                Schnitt = Schnitt ∩ tmp;
                for (j=0; j<n; j++) do
                    if (x_j ≠ free) e_j = l_σ(x_j);
                end;
            end;
        end;
        accepted = accepted ∨ (Schnitt ≠ ∅);
    end;
    return(accepted);
end
```

Abbildung 11: Prüfung der Akzeptanz

Wortes gewahrt bleibt und auch die geratenen Belegungen der freien Variablen konsistent sind. Der Algorithmus ist in Abbildung 11 zu sehen.
Im Kapitel Implementierung wird genauer darauf eingegangen wie sich die Mengen der SD-Zahlen darstellen und manipulieren lassen.
Für die Prüfung ob ein Automat leer ist kann im Grunde das gleiche Verfahren verwendet werden. Es wird so geprüft ob ein Modell existiert dass kleiner oder gleich groß zum geprüften Wort ist. Dieses Vorgehen ist möglich da es zu einer erfüllbaren Presburger Formel ein kleinstes Modell geben muss. Gibt es jedoch

ein Modell der Formel das größer ist als das geprüfte Wort, so wird der Automat falsch als Leer erkannt. Die Länge des zu prüfenden Wortes, und damit die vom Algorithmus ausgeführten Schritte darf also nicht zu klein gewählt werden. Ist n die Länge des Wortes, so ist in der Basis B die größte darstellbare Zahl in diesem Wort $B^n - 1$.
Für die Prüfung der Leerheit müssen aber die Variablen die zuvor durch ein Wort belegt wurden nun wie freie Variablen behandelt und geraten werden. Nur die Platzhalter für Konstanten sind weiterhin zu belegen. Die Länge des Wortes muss hierbei aber nicht unbedingt der Länge der Konstanten entsprechen sondern kann auch länger sein. Ist der Schnitt der durch das Tauschen entstandenen A und Ω Mengen wiederum nicht leer, so ist der Automat nicht leer, da es mindestens eine Belegung und damit ein Wort gibt welches konsistent ist und vom Automaten akzeptiert wird. Der ganze Automat ist entsprechend nur dann leer wenn jeder Zustand q leer ist. Siehe Abbildung 12.
Eine andere Möglichkeit den Automaten auf Leerheit zu prüfen besteht darin den zu diesem SD-Automaten äquivalenten nichtdeterministischen endlichen Automaten zu konstruieren und diesen entsprechend zu prüfen.
Das Verfahren der Schnittmengenbildung ist auch geeignet um ein Modell des Automaten zu erzeugen. Werden wieder alle Variablen als frei angenommen, mit Ausnahme der für die Konstanten benutzen so läßt sich aus der Schnittmenge ein Modell auslesen.

3.1.5 Minimierung

Definition 31 (Minimaler SD-Automat). *Ein Automat $\mathcal{A} = (\mathcal{Q}, \mathcal{D}^{2n}, A, \Omega)$ ist minimal genau dann wenn die drei folgenden Bedingungen erfüllt sind:*

1) $\neg\exists q \in \mathcal{Q}.A(q) = \varnothing$
2) $\neg\exists(q, q') \in \mathcal{Q} \times \mathcal{Q}.q \neq q' \wedge \Omega(q) = \Omega(q')$
3) $\neg\exists q \in \mathcal{Q}.\forall l \in A(q).\exists q' \in \mathcal{Q}.q' \neq q \wedge l \in A(q') \wedge \Omega(q) \subseteq \Omega(q')$

Offensichtlich ist ein Zustand q der nicht erreicht werden kann weil seine Initiale Transition leer ist, $A(q) = \varnothing$, für das Akzeptanzverhalten und die Leerheit des Automaten irrelevant und darf in einem minimalen Automaten nicht vorkommen, was durch die erste Bedingung gewährleistet wird.
Die zweite Bedingung legt fest dass es keine zwei Zustände q, q' gibt die identische finale Transitionen besitzen. q und q' unterscheiden sich lediglich durch ihre initialen Transitionen und akzeptieren danach identische Wörter. Für die Mini-

```
function Empty (A, w, steps)
    m = |w|;
    n = |Variables|;
    empty = true;
    forall (q ∈ Q) do
        for (i=steps; i>=0; i–) do
            if (i>m) do
                for (j=0; j<n; j++) do
                    if (x_j = const) l_σ(x_j) = 0;
                end;
            end;
            l = w_i;
            Schnitt = ∅;
            if (n==m) do
                Schnitt = A(q)[x_0\e_0 ... x_n\e_n];
                for (j=0; j<n; j++) do
                    if (x_j = const) e_j = l_σ(x_j);
                end;
            else
                k = m - i;
                tmp = Ω(q)[x_0\e_(n/2)·k ... x_n\e_(n+n/2)·k];
                Schnitt = Schnitt ∩ tmp;
                for (j=0; j<n; j++) do
                    if (x_j = const) e_j = l_σ(x_j);
                end;
            end;
        end;
        empty = empty ∧ (Schnitt = ∅);
    end;
    return(empty);
end
```

Abbildung 12: Prüfung auf Leerheit

mierung lassen sich diese beiden Zustände zu q'' zusammenfassen mit $A(q'') = A(q) \cup A(q')$ und $\Omega(q'') = \Omega(q)$.
Die letzte Bedingung besagt dass es zu einem Zustand q keine anderen Zustände geben darf die die von der initialen Transition von q akzeptierten Buchstaben l in ihrer initialen Transition enthalten und deren finale Transition die des Zustandes q umfassen. Gibt es zu q solche Zustände, so kann q ohne Auswirkungen auf Akzeptanzverhalten und Leerheit aus der Menge der Zustände $\mathcal{Q}$ gestrichen werden. Werden nicht alle von $A(q)$ akzeptierten l nach den obigen Bedingungen von anderen Zuständen akzeptiert, so lassen sich aus $A(q)$ dennoch diejenigen l streichen die entsprechend von anderen Zuständen akzeptiert werden. Abbildung 13 zeigt den Algorithmus.
Werden zwei minimale Automaten $\mathcal{A}_1$ und $\mathcal{A}_2$ durch eine Konjunktion verknüpft so kann der neu entstandene Automat $\mathcal{A}$ alle drei Bedingungen verletzen. Bei (1) ist das leicht einzusehen. Durch den Schnitt der Transitionen ist es möglich dass die initiale Transition leer wird. Analog läßt sich für (2) argumentieren dass durch den Schnitt zwei Zustände q und q' gleiche finale Transitionen erhalten. Auch kann der Schnitt dazu führen dass $\Omega(q)$ so verkleinert wird dass es ein oder mehrere q' gibt die gegen (3) verstoßen.
Die Disjunktion minimaler Automaten kann Bedingung (1) nicht verletzen, da nur neue Zustände hinzugefügt werden, für die (1) erfüllt ist. Das Hinzufügen von Zuständen kann aber dazu führen dass (2) und (3) nicht länger erfüllt sind.

3.1.6 Übersetzung von Presburger Arithmetik

Da nun beliebige Gleichungen und Ungleichungen in Signed-Digit-Automaten übersetzt werden können muss noch geklärt werden wie mit beliebigen Presburger Formeln verfahren werden muss wenn sie entschieden werden sollen. Es gibt keine Möglichkeit Quantifizierungen zu behandeln, außer der Umwandlung von existenzquantifizierten Variablen in freie Variablen deren Belegungen geraten werden müssen. Allquantoren müssen also in Existenzquantoren transformiert werden. $\forall x.\varphi \equiv \neg\exists x.\neg\varphi$. Dabei treten jedoch Negationen auf die wiederum nicht in SD-Automaten übersetzt werden können.
Problemlos ist die Übersetzung eines existenziellen Fragments der Form

$$\exists x_1 \ldots \exists x_n.\varphi$$

wenn φ keine weiteren Quantoren und keine Negationen enthält. φ enthält dann nur noch Terme die aus Additionen, Subtraktionen und Monomen bestehen, so-

```
function Minimize (Q, A, Ω)
    Q' = ∅;
    A' = ∅;
    Ω' = ∅;
    forall (q ∈ Q) do
        L := A(q);
        forall (q' ∈ Q) do
            if (L == ∅) then
                break;
            end;
            L' := A'(q');
            if (Ω(q) == Ω(q') then
                A' := (A' \ {(q', L')}) ∪ {(q', L' ∪ L)};
                L := ∅;
            else if (Ω(q) ⊂ Ω(q')) then
                L := L \ L';
            else if (Ω(q) ⊃ Ω(q')) then
                A' := A' \ {(q', L')}
                L' := L' \ L;
                if (L' == ∅) then
                    Q' := Q' \ {q'};
                    Ω' := Ω' \ {(q',Ω'(q'))};
                else
                    A' := A' ∪ {(q',L')};
                end;
            end;
            if (L ≠ ∅) then
                Q' := Q' ∪ {q};
                A' := A' ∪ {(q,L)};
                Ω' := Ω' ∪ {(q,Ω(q))};
            end;
        end;
    end;
    return (Q', A', Ω')
end
```

Abbildung 13: Minimierung eines SD-Automaten

wie Relationen zwischen Termen wie $<$, $=$ und $\neq$, oder boolsche Verknüpfungen zwischen Teilformeln.
Die in φ auftretenden Gleichungen können nun in eine der folgenden Formen gebracht werden:

$$\begin{array}{ll} 1) & \sum_{i=0}^{n} c_i \cdot x_i = c \\ 2) & \sum_{i=0}^{n} c_i \cdot x_i < c \\ 3) & \sum_{i=0}^{n} c_i \cdot x_i \neq c \end{array}$$

Diese Gleichungen können mit dem in Abbildung 10 vorgestellten Algorithmus in Automaten übersetzt werden. Da φ auch boolsche Verknüpfungen $\wedge$ und $\vee$ enthalten kann, müssen die für die Gleichungen erzeugten Automaten entsprechend der Gleichung unter Nutzung der Konjunktion und der Disjunktion verknüpft werden.

3.2 Implementierung

Im Rahmen dieser Arbeit wurden die Signed-Digit-Automaten implementiert. Auf die Details dieser Implementierung wird in diesem Kapitel näher eingegangen.
Als Rahmenumgebung für die Implementierung diente dabei das Averest Modellprüfungswerkzeug [14]. Averest ist eine Werkzeugsammlung um synchrone Programme mit gegebenen Eigenschaften zu verifizieren. Dabei kommt als synchrone Sprache $Quartz$ zum Einsatz, die eine Erweiterung von $Esterel$ ist. Der Kompiler $Ruby$ übersetzt das Programm zusammen mit den zu verifizierenden Eigenschaften in die XML Zwischensprache AIF (Averest Interchange Format). Der Modellprüfer $Beryl$ kann dann genutzt werden um die symbolische Modellprüfung durchzuführen. $Beryl$ kann globale, lokale und eine gebundene Modellprüfung durchführen. Die zu prüfenden Eigenschaften werden dabei im μ-Kalkül dargestellt. Für bessere Lesbarkeit gibt es aber auch einen Übersetzer der FairCTL Formeln in den μ-Kalkül übersetzt. Intern benutzt $Beryl$ BDDs und Automaten als Repräsentationsformen für Formeln. Als BDD Pakete können entweder BuDDy [15] oder CUDD [16] gewählt werden. Als weitere Alternative steht das Erfüllbarkeitswerkzeug zChaff [17] zur Verfügung. Als Automatentyp kann bisher zwischen deterministischen endlichen Automaten (DFA) und alternierenden endlichen Automaten (AFA) gewählt werden. Sowohl um die Automatenklassen als auch die aussagenlogischen Klassen gibt es Wrapperklassen um mit einem einheitlichen Interface für eine einfache Austauschbarkeit der verwendeten Klassen zu sorgen. Die neue Automatenklasse der Signed Digit Automata (SDA) muss sich entsprechend an dieses Interface halten.
Entscheidend für die Effizienz der Klasse ist die Repräsentation und Manipulation von Mengen von Signed Digits. Da $Beryl$ ein symbolisch arbeitendes Werkzeug ist empfiehlt es sich diese Mengen ebenfalls symbolisch zu repräsentieren. Aus diesem Grund werden die Mengen durch BDDs repräsentiert. Dazu werden zuächst die Signed Digits durch Zweier-Komplement Zahlen dargestellt und anschließend BDDs generiert die genau dann zu $true$ ausgewertet werden wenn die Belegung der BDD-Variablen dem Bitvektor einer Zweier-Komplement-Zahl der Signed-Digit-Menge entspricht.
Es ist offensichtlich dass eine Variable des Automaten mehrere BDD Variablen zu ihrer Darstellung benötigt. Da für die verwendeten Signed Digit Zahlen eine Basis B gewählt werden muss so dass für jede Ziffer a gilt dass $B < a < B$, ergibt sich dass für jede Ziffer $n = \lceil log_2(2 \cdot B) \rceil + 1$ BDD-Variablen gebraucht werden.
Die Klasse der Signed Digit Mengen stellt Konstruktoren zur Verfügung die zu bestimmten Prädikaten die entsprechenden Mengen konstruieren. Darüber hinaus

Operatoren um Schnitt- und Vereinigungsmengen zu bilden oder die Leerheit der Menge zu prüfen.
In den nun folgenden Abschnitten werden die verwendeten Methoden genauer vorgestellt.

3.2.1 Mengenkonstruktion für Prädikate

Am einfachsten ist die Konstruktion des BDDs für das Prädikat $EQ0$, da initiale und finale Transition aus genau einem einzigen Signed Digit bestehen:
$A(q) = \Omega(q) = \{(d_0, \ldots, d_{2n-1}) \mid d_{\sigma(x)} = 0\}$
Beide BDDs sind somit identisch und evaluieren genau dann zu $true$ wenn alle BDD-Variablen die der Variable x zugeordnet wurden mit 0 belegt wurden.
Sei dazu $\mathcal{V}_1$ die Menge der Automatenvariablen, und $\mathcal{V}_2$ die Menge der BDD-Variablen, und weiter $\iota : \mathcal{V}_1 \rightarrow \mathcal{V}_2^n$ eine Funktion die einer Automatenvariable ihre BDD-Variablen zuordnet.
Dann ist der zu konstruierende BDD äquivalent zu $bdd = \neg\iota(x)_0 \wedge \ldots \wedge \neg\iota(x)_{n-1}$
Ähnlich einfach ist die Umsetzung des Prädikats $LT0C$. Hier ist die initiale Transition $A(q) = \{(d_0, \ldots, d_{2n-1}) \mid d_{\sigma(x)} < 0\}$. Der BDD der dazu erzeugt werden muss wird wahr wenn die Variablenbelegung einer negativen Zahl entspricht. Da die Signed Digits durch 2k Zahlen dargestellt werden ist die Zahl genau dann negativ wenn das höchstwertige Bit 1 ist. Also ist der BDD äquivalent zu $bdd = \iota(x)_{n-1}$
Bei der finalen Transition ist die Mengenkonstruktion etwas schwieriger. Sie ist allgemein definiert durch $\Omega(q) = \mathcal{D}^{2n}$. Hier ergibt sich durch die Verwendung der Zweierkomplementdarstellung eine Schwierigkeit. Die Definition von Ω besagt dass alle Signed Digits von q akzeptiert werden. Dies kann jedoch nicht in den BDD für $bdd = true$ übersetzt werden. Sei als ein Beispiel $\mathcal{B} = 3$. dann ist $n = 3$. Damit lassen sich die Zahlen von -4 bis 3 darstellen, und diese würden auch akzeptiert wenn der BDD für $bdd = true$ gewählt würde. Stattdessen ist zu $-B < x < B$ die Menge der Signed Digits zu konstruieren. Betrachtet man die initiale Transition nochmals, so zeigt sich auch dort dass die konstruierte Menge diese Schwachstelle aufweist. Die Menge repräsentiert alle Signed Digits die $x < 0$ erfüllen. Korrekt wäre aber die Menge $-B < x < 0$.
Um einen BDD zu erzeugen der $-B < x < B$ entspricht ist größerer Aufwand zu betreiben. Die Gleichung kann zerlegt werden in $-B < x \wedge x < B$, was wiederum identisch ist mit $0 \leq x+B+1 \wedge x-B < 0$. Die Prüfungen eines Bitvektors auf ≥ 0 und < 0 sind einfach da in beiden Fällen nur das höchstwertige Bit geprüft werden muss. Im Falle von $y < 0$ muss $y_{n-1} = 1$ sein, im Falle $y \geq 0$ muss $y_{n-1} = 0$ sein.

```
function lengthen (x, y)
    if (|x| < |y|) do
        n = |x|;
        d = |y| - |x|;
        for (i = 0, i < d, i++) do
            append (x, x_{n-1});
        end;
    end;
    return (x, y)
end
```

Abbildung 14: Hilfsfunktion zur Angleichung der Länge

Nun liegen die Bitvektoren $x + B + 1$ und $x - B$ noch nicht vor, sondern müssen zuerst erzeugt werden. Dazu kann die Addition von Zweierkomplement Zahlen verwendet werden. Es muss aber beachtet werden dass die Belegung von x zum Zeitpunkt der Konstruktion nicht bekannt ist sondern die ganze Menge Signed Digits die das Prädikat erfüllen den BDD zu $true$ evaluieren sollen. Die Bitvektoren der Summen $x + B + 1$ und $x - B$ hängen also weiterhin von den BDD-Variablen $\iota(x)$ ab. Es liegen also keine Bitvektoren vor aber es lassen sich Vektoren von BDDs erzeugen die das entsprechende Bit abhängig von x bzw. $\iota(x)$ erzeugen. Da für die Mengenkonstruktion der anderen Prädikate ebenfalls Operationen auf solchen BDD-Vektoren nötig sind empfiehlt es sich die benötigten Operationen in Hilfsfunktionen auszulagern. Diese Hilfsfunktionen sind in den Abbildungen 14 bis 21 zu sehen. Für diese Hilfsfunktionen ist eine notwendig die gewährleistet dass die übergebenen BDD-Vektoren die gleiche Länge besitzen. Dies ist durch die Vorzeichenerweiterung der Zweier-Komplement-Zahlen problemlos möglich. Es muss lediglich das höchstwertige Bit kopiert werden. Die Funktion **lengthen** erhält zwei BDD-Vektoren und erweitert den Ersten so dass er nicht kürzer als der Zweite ist.
Es müssen auch zwei BDD-Vektoren addiert werden. Diese Aufgabe übernimmt die Funktion **add**. Sie realisiert einen Volladdierer der durch eine Schleife die Summe der beiden Vektoren bestimmt. Die jeweils errechneten BDDs, welche das Summenbit repräsentieren werden zu einem neuen Vektor zusammengefasst. Dabei wird das letzte Übertragsbit ignoriert. Es muss also darauf geachtet werden dass kein Übertrag stattfinden kann. An einem Beispiel kann man sehen dass das Übertragsbit ignoriert werden muss, da es sonst zu falschen Ergebnissen kommen

```
function add (x, y)
    x = lengthen (x, y);
    y = lengthen (y, x);
    bddvector z;
    bdd c, s;
    c = bdd(true);

    for (i = 0, i < |x|, i++) do
        s := x_i xor y_i xor c;
        c := y_i ∧ c ∨ (x_i ∧ c) ∨ (x_i ∧ y_i);
        append (z, s);
    end;
    return (z)
end
```

Abbildung 15: Hilfsfunktion für Addition

kann.

$$\begin{array}{cccc} & 1 & 0 & 1 \\ + & 0 & 1 & 1 \\ \hline 1 & 0 & 0 & 0 \end{array} \qquad \begin{array}{cc} & -3 \\ + & 3 \\ \hline & 0 \end{array}$$

Andererseits kann es zu Problemen kommen wenn die BDD-Vektoren zu denen die Summe gebildet werden soll zu kurz sind, wie das folgende Beispiel zeigt.

$$\begin{array}{cccc} & 0 & 1 & 1 \\ + & 0 & 1 & 1 \\ \hline & 1 & 1 & 0 \end{array} \qquad \begin{array}{cc} & 3 \\ + & 3 \\ \hline & 6 \end{array}$$

Hier wird die Summe zwar korrekt bestimmt, jedoch ist die Interpretation nicht die 6 sondern die -2. Um dies zu vermeiden müssen die BDD-Vektoren die addiert werden für jede Addition an der sie teilhaben um eine Stelle erweitert werden.
Subtraktionen lassen sich auf Additionen zurückführen. Dazu wird der Subtrahend x dabei durch $-x$ ersetzt, was sich leicht durch $\neg x + 1$ bestimmen läßt. Die

```
function invert (x)
        bddvector z;

        for (i = 0, i < |x|, i++) do
                append (z, ¬x_i);
        end;
        return (z)
end
```

Abbildung 16: Hilfsfunktion zur Invertierung

```
function equals (x, y)
        x = lengthen (x, y);
        y = lengthen (y, x);
        bdd s;
        s = bdd(true);

        for (i = 0, i < |x|, i++) do
                s := s ∧ (x_i ∧ y_i ∨ ¬x_i ∧ ¬y_i);
        end;
        return (s)
end
```

Abbildung 17: Hilfsfunktion für Gleichheit

dafür nötige Addition ist schon vorgestellt, es wird noch eine Hilfsfunktion für die Negation benötigt.
Der Vergleich zweier Vektoren auf Gleichheit wird später ebenfalls benötigt. Die Funktion **equals** prüft die Äquivalenz der einzelnen BDDs der Vektoren x und y und gibt einen BDD zurück der besagt ob alle BDDs x_i und y_i gleich sind. Die Vergleiche ob die von einem BDD-Vektor repräsentierte Zahl kleiner oder größer 0 ist lassen sich anhand des höchstwertigen Bits des Vektors entscheiden. Ist der BDD dieses Bits $true$ so ist die Zahl negativ, anderenfalls ist die Zahl größer oder gleich 0. Die Funktionen **lt0** und **ge0** sind also recht kurz.
Eine weitere benötigte Hilfsfunktion ist die Umwandlung eines Integers in eine Zweier-Komplement Darstellung, wie es **conv** erledigt. Die Menge der benötigten

```
function lt0 (x)
    n = |x|;
    return (x_{n-1})
end

function ge0 (x)
    n = |x|;
    return (¬x_{n-1})
end
```

Abbildung 18: Hilfsfunktionen zum Vergleich mit 0

BDDs n wird hier um 2 erhöht, um zu gewährleisten dass die Vektoren nicht zu kurz sind und es so nicht zu Fehlern kommen kann.
Für die Konstruktionen der Prädikate für Addition, Subtraktion und Gleichheit wird auf eine Funktion $t(x, y)$ zurückgegriffen.

$$t(x,y) := \begin{cases} +1 & : \text{falls } x + y \geq \mathcal{A} \\ -1 & : \text{falls } x + y \leq \text{-}\mathcal{A} \\ 0 & : \text{sonst} \end{cases}$$

Dazu läßt sich eine Hilfsfunktion konstruieren, der zwei BDDs a, b und drei BDD-Vektoren x, y, z übergeben werden. Zurückgegeben werden soll ein BDD-Vektor der abhängig von den beiden BDDs a und b einem der drei Vektoren x, y oder z entspricht. Im Falle von $a \wedge \neg b$ soll das Ergebnis x, im Falle von $\neg a \wedge b$ soll das Ergebnis y entsprechen. Ist $\neg a \wedge \neg b$ erfüllt muss z ds Ergebniss sein. Da während der Konstruktion die Wahrheit der BDDs nicht bekannt ist muss der zurückgegebene Vektor t die Entscheidung mit enthalten. Dies läßt sich realisieren indem t Bitweise so erzeugt wird dass a und b das Ergebnis t_i auswählen. Aussagenlogisch formuliert ist das in der Formel: $t_i = a \wedge \neg b \wedge x_i \vee \neg a \wedge b \wedge y_i \vee \neg a \wedge \neg b \wedge z_i$. Die Funktion **select** konstruiert auf genau diese Weise die einzelnen BDDs des Rückgabe-Vektors.
Mit den vorgestellten Hilfsfunktionen lassen sich nun BDDs konstruieren die den Signed-Digit-Mengen der Transitionen der Automaten entsprechen. Der BDD welcher für die finale Transition des Prädikats $LT0C$ gewährleistet dass $-B < x < B$ gilt wird unter Verwendung der Hilfsfunktionen durch **valid** erzeugt. Die Verwendung von **valid** ist auch in der Konstruktion für andere Prädikate nützlich, um Fehlern durch das Raten der freien Variablen vorzubeugen.

```
function conv (num)
    bddvector z;
    n = log2(2 · B) + 2;

    if (|num| ≤ B) then
        if (num < 0) do
            num = (-1 · num) - 1;
            for (i = 0; i < n; i++) do
                if (num mod 2 == 1) c = bdd(false)
                else c = bdd(true)
                append (z, c);
                num = num / 2;
            end;
        end;
        if (num ≥ 0) do
            for (i = 0; i < n; i++) do
                if (num mod 2 == 0) c = bdd(false)
                else c = bdd(true)
                append (z, c);
                num = num / 2;
            end;
        end;
    end;
    return (z)
end
```

Abbildung 19: Hilfsfunktion zur Konvertierung eines Integers

```
function select (a, b, x, y, z)
    bddvector t;
    bdd c;
    x = lengthen (x, y);
    x = lengthen (x, z);
    y = lengthen (y, x);
    z = lengthen (z, x);

    for (i = 0; i < |x|; i++) do
        c = ¬a ∧ ¬b ∧ z_i;
        c = c ∨ (¬a ∧ b ∧ y_i);
        c = c ∨ (a ∧ ¬b ∧ x_i);
                append (t, c);
    end;
    return (t);
end
```

Abbildung 20: Hilfsfunktion zur Selection

```
function valid (x)
    bdd z;

    aneg = conv (-1 · (B - 1));
    a = conv (B - 1);
    oneneg = conv (-1);
    one = conv (1);
    xaneg = add (x, aneg);
    xanegsubone = add (xaneg, oneneg);
    notbigger = lt0 (xanegsubone);
    xa = add (x, a);
    notsmaller = ge0 (xa);
    z = notbigger ∧ notsmaller;
    return (z);
end
```

Abbildung 21: Konstruktion für $-B < x < B$

function Construct-ADDC (x, y, z, x', y', $alpha$)
$digits$ = **valid** (x) $\wedge$ **valid** (y) $\wedge$ **valid** (z);
$digits = digits \wedge$ **valid** (x') $\wedge$ **valid** (y') $\wedge$ **valid** (z');

$aneg$ = **conv** (-1 $\cdot$ ($\mathcal{B}$ - 1));
a = **conv** ($\mathcal{B}$ - 1);
$bneg$ = **conv** (-1 $\cdot$ $\mathcal{B}$);
b = **conv** ($\mathcal{B}$);
$oneneg$ = **conv** (-1);
one = **conv** (1);
$zero$ = **conv** (0);
xy = **add** (x, y);
$xyaneg$ = **add** (xy, $aneg$);
xya = **add** (xy, a);
$xyasubone$ = **add** (xya, $oneneg$);
$xylea$ = **lt0** ($xyasubone$);
$xygea$ = **ge0** ($xyaneg$);
$x'y'$ = **add** (x', y');
$x'y'aneg$ = **add** ($x'y'$, $aneg$);
$x'y'a$ = **add** ($x'y*$, a);
$x'y'asubone$ = **add** ($x'y'a$, $oneneg$);
$x'y'lea$ = **lt0** ($x'y'asubone$);
$x'y'gea$ = **ge0** ($x'y'aneg$);

t = **select** ($x'y'gea$, $x'y'lea$, one, $oneneg$, $zero$);
$tnegb$ = **select** ($xygea$, $xylea$, $bneg$, b, $zero$);
$s1$ = **add** (xy, t);
$s2$ = **add** ($s1$, $tnegb$);
$s2eqz$ = **equals** ($s2$, z);
$t2$ = **select** ($xygea$, $xylea$, one, $oneneg$, $zero$);
i = **equals** ($t2$, $zero$);
$digits = digits \wedge s2eqz$;
if ($alpha$) $digits = digits \wedge i$;
return ($digits$);
end

Abbildung 22: Konstruktion von ADDC

function Construct-SUBC (x, y, z, x', y', $alpha$)
$digits$ = **valid** (x) $\wedge$ **valid** (y) $\wedge$ **valid** (z);
$digits = digits \wedge$ **valid** (x') $\wedge$ **valid** (y') $\wedge$ **valid** (z');
$aneg$ = **conv** (-1 $\cdot$ ($\mathcal{B}$ - 1));
a = **conv** ($\mathcal{B}$ - 1);
$bneg$ = **conv** (-1 $\cdot$ $\mathcal{B}$);
b = **conv** ($\mathcal{B}$);
$oneneg$ = **conv** (-1);
one = **conv** (1);
$zero$ = **conv** (0);
$ynot$ = **invert** (y);
$yneg$ = **add** ($ynot$, one);
$y'not$ = **invert** (y');
$y'neg$ = **add** ($y'not$, one);
xy = **add** (x, $yneg$);
$xyaneg$ = **add** (xy, $aneg$);
xya = **add** (xy, a);
$xyasubone$ = **add** (xya, $oneneg$);
$xylea$ = **lt0** ($xyasubone$);
$xygea$ = **ge0** ($xyaneg$);
$x'y'$ = **add** (x', $y'neg$);
$x'y'aneg$ = **add** ($x'y'$, $aneg$);
$x'y'a$ = **add** ($x'y*$, a);
$x'y'asubone$ = **add** ($x'y'a$, $oneneg$);
$x'y'lea$ = **lt0** ($x'y'asubone$);
$x'y'gea$ = **ge0** ($x'y'aneg$);
t = **select** ($x'y'gea$, $x'y'lea$, one, $oneneg$, $zero$);
$tnegb$ = **select** ($xygea$, $xylea$, $bneg$, b, $zero$);
$s1$ = **add** (xy, t);
$s2$ = **add** ($s1$, $tnegb$);
$s2eqz$ = **equals** ($s2$, z);
$t2$ = **select** ($xygea$, $xylea$, one, $oneneg$, $zero$);
i = **equals** ($t2$, $zero$);
$digits = digits \wedge s2eqz$;
if ($alpha$) $digits = digits \wedge i$;
return ($digits$);
end

Abbildung 23: Konstruktion von SUBC

function Construct-EQ (x, y, x', y', $alpha$)
$digits$ = **valid** (x) $\wedge$ **valid** (y) $\wedge$ **valid** (z);
$digits = digits \wedge$ **valid** (x') $\wedge$ **valid** (y') $\wedge$ **valid** (z');
$aneg$ = **conv** (-1 $\cdot$ ($\mathcal{B}$ - 1));
a = **conv** ($\mathcal{B}$ - 1);
$bneg$ = **conv** (-1 $\cdot$ $\mathcal{B}$);
b = **conv** ($\mathcal{B}$);
$oneneg$ = **conv** (-1);
one = **conv** (1);
$zero$ = **conv** (0);
$ynot$ = **invert** (y);
$yneg$ = **add** ($ynot$, one);
$y'not$ = **invert** (y');
$y'neg$ = **add** ($y'not$, one);
xy = **add** (x, $yneg$);
$xyaneg$ = **add** (xy, $aneg$);
xya = **add** (xy, a);
$xyasubone$ = **add** (xya, $oneneg$);
$xylea$ = **lt0** ($xyasubone$);
$xygea$ = **ge0** ($xyaneg$);
$x'y'$ = **add** (x', $y'neg$);
$x'y'aneg$ = **add** ($x'y'$, $aneg$);
$x'y'a$ = **add** ($x'y*$, a);
$x'y'asubone$ = **add** ($x'y'a$, $oneneg$);
$x'y'lea$ = **lt0** ($x'y'asubone$);
$x'y'gea$ = **ge0** ($x'y'aneg$);
t = **select** ($x'y'gea$, $x'y'lea$, one, $oneneg$, $zero$);
$tnegb$ = **select** ($xygea$, $xylea$, $bneg$, b, $zero$);
$s1$ = **add** (xy, t);
$s2$ = **add** ($s1$, $tnegb$);
$s2eq0$ = **equals** ($s2$, $zero$);
$t2$ = **select** ($xygea$, $xylea$, one, $oneneg$, $zero$);
i = **equals** ($t2$, $zero$);
$digits = digits \wedge s2eqz$;
if ($alpha$) $digits = digits \wedge i$;
return ($digits$);
end

Abbildung 24: Konstruktion von EQ

Für das Prädikat $ADDC$ wird ein Zustand q erzeugt dessen initiale und finale Transitionen durch die folgenden Signed-Digit-Mengen gegeben sind:
$A(q) = \mathcal{S} \cap \mathcal{I} und \Omega(q) = \mathcal{S}$,
wobei $\mathcal{S} = \{(d_0, \ldots, d_{2n-1}) \mid s(d_{\sigma(x)}, d_{\sigma(y)}, d_{\sigma(x)+n}, d_{\sigma(y)+n}) = d_{\sigma(z)}\}$ und $\mathcal{I} = \{(d_0, \ldots, d_{2n-1}) \mid t(d_{\sigma(x)}, d_{\sigma(y)}) = 0\}$.
Die Konstruktion sowohl der initialen als auch der finalen Transition erledigt die Funktion **Construct-ADDC**, wie sie in Abbildung 22 zu sehen ist. Das Vorgehen ist durch die vorher definierten Funktionen einfach. Zunächst werden mit **conv** BDD-Vektoren zu Konstanten erzeugt. Danach müssen BDDs a und b erzeugt werden die zur Verwendung in **select** benutzt werden können und die Ungleichungen $x+y \geq B$ und $x+y \leq B$ bzw $x'+y' \geq B$ und $x'+y' \leq B$ beschreiben. für $t(x', y')$ werden **select** die BDD-Vektoren für $+1$, -1 und 0 übergeben, während für $-t(x, y) \cdot B$ die BDD-Vektoren für $-B$, $+B$ und 0 übergeben werden. Im nächsten Schritt wird die Summe berechnet und auf Gleichheit mit z überprüft. Durch $alpha$ kann bei der Konstruktion berüccksichtigtksichtigt werden ob der erzeugte BDD für die initiale oder die finale Transition verwendet wird und gegebenenfallls noch der Schnitt mit $\mathcal{I}$ durchgeführt werden.
Wie schon zuvor erwähnt kann die kann die Subtraktion auf die Addition zurückgeführt werden. Die Mengen-Konstruktion für $SUBC$ ist identisch zu der für $ADDC$ mit dem einzigen Unterschied dass y und y' zu $-y$ und $-y'$ konvertiert werden müssen.
Die Konstruktion für das Prädikat EQ ist ebenfalls einfach. Die Definition lautet: $A(q) = \mathcal{S} \cap \mathcal{I}$ und $\Omega(q) = \mathcal{S}$, wobei $\mathcal{S} = \{ (d_0, \ldots, d_{2n-1}) \mid s(d_{\sigma(x)}, -d_{\sigma(y)}, d_{\sigma(x)+n}, -d_{\sigma(y)+n}) = 0\}$ und $\mathcal{I} = \{ (d_0, \ldots, d_{2n-1}) \mid t(d_{\sigma(x)}, d_{\sigma(y)}) = 0 \}$.
Wie bei der Subtraktion werden also y und y' zunächst zu $-y$ und $-y'$ konvertiert. Nach der Berechnung der Summe wird entsprechend mit 0 statt mit z verglichen.

3.2.2 Konstruktion der Prädikate

Die Automatenklasse selbst besteht nun aus zwei Listen für die BDDs die zu jedem vorhandenen Zustand die BDDs für A und Ω speichern. Weiter wird noch eine Liste $\mathcal{P}$ benötigt in der die Funktion **assign-const** Paare von Variablen und den ihnen zugewiesenen Werten ablegen kann, und mit der bei der Prüfung der Akzeptanz oder der Leerheit das Wort erweitert, bzw. erzeugt werden kann.
Die Automatenkonstruktion für die Prädikate besteht also lediglich darin die dem Prädikat entsprechenden BDDs zu erzeugen und in den Listen zu speichern. Da die Automaten der Prädikate aus einem einzigen Zustand bestehen enthalten die Listen auch nur je einen BDD.

Für die Prädikate ADD, SUB, $LT0$, LT und MON werden entsprechend ihren Definitionen keine BDDs konstruiert, sondern neue Variablen angelegt und die benötigten Prädikate erzeugt und verknüpft.
Im Falle der Addition ADD sind diese Prädikate $ADDC$ und EQ, die Verknüpfung ist $ADD(x, y, z) := ADDC(x, y, u) \wedge EQ(u, z)$. Die Variable u muss dabei neu angelegt werden.
Das Anlegen von Variablen übernimmt die Klasse der Signed-Digit-Mengen. Dort werden durch die Funktion **new-var** entsprechend der Basis B neue BDD-Variablen im verwendeten BDD-Paket angelegt und mit der neuen Automaten-Variable assoziiert.

3.2.3 Operationen auf Automaten

Die Verwendung von BDDs zur Darstellung der Transitionen hat auch Auswirkungen auf die Algorithmen die auf den Automaten operieren. Betrachtet man die Konjunktion zweier Automaten $\mathcal{A}_1 \wedge \mathcal{A}_2 := (\mathcal{Q}_1 \times \mathcal{Q}_2, \mathcal{D}^{2n}, A_{12}, \Omega_{12})$ mit $A_{12}(q_1, q_2) = A_1(q_1) \cap A_2(q_2)$ und $\Omega_{12}(q_1, q_2) = \Omega_1(q_1) \cap \Omega_2(q_2)$, so zeigt sich dass die notwendigen Schnitte für A_{12} und Ω_{12} durch Konjuktionen der BDDs zu $A_1(q_1)$ und $A_2(q_2)$ bzw. $\Omega_1(q_1)$ und $\Omega_2(q_2)$ vorgenommen werden können. Durch die Konjunktion der BDDs schränkt sich die Menge der Modelle auf diejenigen ein die Modelle beider BDDs sind und damit repräsentiert der BDD der Konjunktion nur noch die Signed-Digit-Zahlen die im Schnitt der Transitionen liegen. Der Algorithmus für die Konjunktion ist Abbildung 25 zu sehen. Die Disjunktion zweier Automaten $\mathcal{A}_1 \vee \mathcal{A}_2 := (\mathcal{Q}_1 \cup \mathcal{Q}_2, \mathcal{D}^{2n}, A_1 \cup A_2, \Omega_1 \cup \Omega_2)$ ist unabhängig von der Art der Darstellung der Transitionen da lediglich die Mengen der Transitionen und Zustände vereinigt werden, was keine Änderung an den Signed-Digit-Mengen bewirkt. Der Algorithmus ist in Abbildung 26 gegeben.
Bei beiden Verknüpfungen der Automaten muss auch gewährleistet sein dass im erzeugten Automaten die Listen der Platzhalter für Konstanten $\mathcal{P}$ vorhanden sind. Es gilt unabhängig davon ob eine Konjunktion oder eine Disjunktion vorgenommen wird werden diese Listen verkettet.
Interessanter ist die Betrachtung der Algorithmen zur Prüfung auf Akzeptanz eines Wortes und der Prüfung auf Leerheit. In beiden Algorithmen werden Methoden benötigt um in einer Menge von Signed-Digits die benutzten Variablen zu tauschen um die Konsistenz des akzeptierten Wortes zu gewährleisten. Durch die Verwendung der BDD-Pakete steht eine Funktion **swap** zur Verfügung die zwei Mengen an BDD-Variablen in einem bestimmten BDD gegeneinander austauscht. Sollen also zwei Variablen x und y getauscht werden genügt es die Funktion **swap**

```
function AND (A_1, A_2)
    Q = ∅;
    A = ∅;
    Ω = ∅;
    forall (q_1 in Q_1) do
        forall (q_2 in Q_2) do
            Q = Q ∪ (q_1, q_2);
            A = A ∪ A(q_1) ∧ A(q_2);
            Ω = Ω ∪ Ω(q_1) ∧ Ω(q_2);
        end;
    end;
    P = P_1 ∪ P_2;
    return (Q, D^{2n}, A, Ω, P);
end
```

Abbildung 25: Konjunktion

mit den Listen von BDD-Variablen für x und y, $\iota(x)$ und $\iota(y)$ aufzurufen. Diese Vorgehen ist leicht erweiterbar um Ersetzungen der Art $A(q)[x_0 \setminus e_0 \ldots x_n \setminus e_n]$ in einem Aufruf durchführen zu können. Es sind einfach die Listen der BDD-Variablen für alle zu tauschenden Variablen aneinander zu hängen.
In beiden Automatenoperationen müssen vorhandene Variablen auch mit den dem Wort entsprechenden Werten belegt werden. Entweder alle Variablen die vom Wort benutzt werden, oder nur solche die eine Konstante repräsentieren. Das Vorgehen ist in beiden Fällen identisch. Eine Variable x des Automaten wird durch eine Signed-Digit-Ziffer a belegt in dem die BDD-Variablen $\iota(x)$ mit der Zweier-Komplement Repäsentation der Ziffer $\eta(a)$ belegt werden. $\eta(a)$ läßt sich analog zur Hilfsfunktion **conv** in Abbildung 19 bestimmen, mit dem Unterschied dass η kein Vektor von BDDs ist, sondern von boolschen Werten. Die Funktion **assign** in Abbildung 27 bestimmt einen BDD in dem die Variable x mit dem Wert a belegt wird in dem die Variablen $\iota(x)$ entsprechend $\eta(a)$ negiert oder nicht negiert mit dem BDD b verknüpft werden. Die Abbildungen 28 und 29 zeigen die Algorithmen **Accept** und **Empty** unter Verwendung von BDDs.

```
function OR (A_1, A_2)
    Q = ∅;
    A = ∅;
    Ω = ∅;
    forall (q_1 in Q_1) do
        Q = Q ∪ {q_1};
        A = A ∪ A(q_1);
        Ω = Ω ∪ Ω(q_1);
    end;
    forall (q_2 in Q_2) do
        Q = Q ∪ {q_2};
        A = A ∪ A(q_2);
        Ω = Ω ∪ Ω(q_2);
    end;
    P = P_1 ∪ P_2;
    return (Q, D^{2n}, A, Ω, P);
end
```

Abbildung 26: Disjunktion

```
function assign (b, x, a)
    assigned = bdd(true);
    for (i=0; i<|ι(x)|; i++) do
        if (η_i(a)) then assigned = assigned ∧ ι_i(x);
        else assigned = assigned ∧ ¬ι_i(x);
    end;
    assigned = b ∧ assigned;
    return (assigned);
end
```

Abbildung 27: Variablenbelegung

```
function Accept (A, w)
    m = |w|;
    n = |Variables|;
    accepted = false;
    forall (q ∈ Q) do
        for (i=m; i>=0; i–) do
            l = w_i;
            bdd S;
            if (n==m) do
                s_1 = {x_0 ... x_n};
                s_1 = {e_0 ... e_n};
                S = swap(A(q), s_1, s_2);
                for (j=0; j<n; j++) do
                    if (x_j ≠ free) S=assign(S, e_j, l_σ(x_j));
                end;
            else
                k = m - i;
                s_1 = {x_0 ... x_n};
                s_1 = {e_(n/2)·k ... e_(n+n/2)·k};
                tmp = swap(Ω(q), s_1, s_2);
                S = S ∧ tmp;
                for (j=0; j<n; j++) do
                    if (x_j ≠ free) S=assign(S, e_j, l_σ(x_j));
                end;
            end;
        end;
        accepted = accepted ∨ (S ≠ ∅);
    end;
    return(accepted);
end
```

Abbildung 28: Prüfung der Akzeptanz

```
function Empty (A, w, steps)
    m = |w|;
    n = |Variables|;
    empty = true;
    forall (q ∈ Q) do
        for (i=steps; i>=0; i–) do
            if (i>m) do
                for (j=0; j<n; j++) do
                    if (x_j = const) l_σ(x_j) = 0;
                end;
            end;
            l = w_i;
            bdd S;
            if (n==m) do
                s_1 = {x_0 ... x_n};
                s_1 = {e_0 ... e_n};
                S = swap(A(q), s_1, s_2);
                for (j=0; j<n; j++) do
                    if (x_j ≠ free) S = assign(S, e_j, l_σ(x_j));
                end;
            else
                k = m - i;
                s_1 = {x_0 ... x_n};
                s_1 = {e_(n/2)·k ... e_(n+n/2)·k};
                tmp = swap(Ω(q), s_1, s_2);
                S = S ∧ tmp;
                for (j=0; j<n; j++) do
                    if (x_j ≠ free) S=assign(S, e_j, l_σ(x_j));
                end;
            end;
        end;
        empty = empty ∧ (Schnitt = ∅);
    end;
    return(empty);
end
```

Abbildung 29: Prüfung auf Leerheit

3.3 Laufzeitergebnisse

Im Rahmen dieser Arbeit lag das Augenmerk auch auf der Beurteilung der Effizienz der im Vorangegangenen vorgestellten Algorithmen. Dazu wurden zunächst die Zeiten für die Konstruktion der Prädikate gemessen und die Größen der BDDs der initialen und der finalen Transition betrachtet. Danach wurden die erstellten Automaten auf Leerheit in 10 und 5 Schritten geprüft. Diese Tests wurden auf einem Xeon Rechner mit zwei Prozessoren mit 3,06 GHz und 4GB Arbeitsspeicher durchgeführt. Als BDD Paket wurde für die endlichen Automaten und die Signed-Digit-Automaten CUDD verwendet. Für die alternierenden Automaten wurde zChaff verwendet.
Die nachfolgenden Tabellen zeigen die bei den Tests gemessenen Zeiten für die Konstruktion der Prädikate, für die Konstruktion von Automaten zu Monomen und schließlich für Gleichungen verschiedener Größen. Alle erzeugten Automaten bestehen aus nur einem Zustand, zu dessen initialer und finaler Transition die BDD-Größen, d.h. die Menge von Knoten in diesen BDDs, angegeben sind. Die Signed-Digit-Automaten wurden einmal mit der Basis $B = 3$ und einmal mit der Basis $B = 7$ konstruiert. Zusätzlich wurde zu den konstruierten Automaten die Anzahl der im Automaten benutzten Variablen notiert. Zum Vergleich wurden die Gleichungen auch in endliche Automaten und in alternierende Automaten übersetzt.
Die gemessenen Zeiten wurden in Sekunden angegeben. Traten während der Messungen Zeiten von mehr als 10 Minuten auf, so wurden die gerade durchgeführten Operationen abgebrochen.

Prädikat	Vars	A	Ω	Konstruktion	Leerheit 5	Leerheit 10
EQ0	1	4	4	0	0	0
LT0C	1	3	5	0	0	0
LT0	2	63	108	0	0,01	0,01
EQ	2	115	108	0	0,01	0,01
LT	3	290	533	0,01	0,18	0,25
ADDC	3	266	281	0,01	0,01	0,03
ADD	4	469	533	0,02	0,19	0,26
SUBC	3	266	281	0,01	0,01	0,02
SUB	4	469	533	0,01	0,18	0,27

Abbildung 30: Konstruktion der Prädikate in der Basis 3

Monome	Vars	A	Ω	Konstruktion	Leerheit 5	Leerheit 10
1·x = y	6	533	526	0,06	2,53	177,34
2·x = y	8	473	1292	0,14	16,71	> 10 min
3·x = y	10	3903	11674	0,53	> 10 min	> 10 min
4·x = y	10	1755	1823	0,2	> 10 min	> 10 min
5·x = y	12	5555	10052	0,51	> 10 min	> 10 min
10·x = y	14	4285	11812	14,17	> 10 min	> 10 min
30·x = y	20	6712	11148	55,84	> 10 min	> 10 min
300·x = y	28	122339	301809	1312,03	> 10 min	> 10 min

Abbildung 31: Konstruktion von Monomen in der Basis 3

Terme	Vars	A	Ω	Konstruktion	Leerheit 5	Leerheit 10
1·x = c	6	469	511	0,1	2,21	> 10 min
2·x = c	8	625	1438	0,15	3,5	> 10 min
3·x = c	10	27974	66334	9,45	> 10 min	> 10 min
4·x = c	10	740	1707	0,3	10,17	> 10 min
5·x = c	12	32651	55708	5,59	> 10 min	> 10 min
6·x = c	12	7537	35552	23,6	>10 min	> 10 min
10·x = c	14	4440	12096	14,09	> 10 min	> 10 min
30·x = c	20	6861	11309	55,93	> 10 min	> 10 min
300·x = c	28	122498	302261	1314,78	> 10 min	> 10 min

Abbildung 32: Konstruktion von Gleichungen in der Basis 3

Vergleicht man zunächst die Konstruktion für die Basis 3 mit der für die Basis 8 so sieht man dass die konstruierten BDDs zur Basis 8 immer größer sind als die zur Basis 3. Dies ist nicht verwunderlich da mehr BDD-Variablen vorhanden sind und die in den erzeugten BDDs verwendet werden.
Unabhänig von der verwendeten Basis sieht man dass mit der Größe der BDDs auch die Konstruktionszeiten steigen. Eine nähere Betrachtung hat gezeigt, dass das meiste der gemessenen Zeit in den BDD-Paketen verbracht wird. Die Größen der BDDs sind also das hauptsächliche Merkmal der Effizienz der Algorithmen.
Bei der Konstruktion der Automaten zu Monomen $a \cdot x = y$ zeigt sich dass größere Faktoren a nicht notwendig zu größeren BDDs führen. Jedoch steigen im Allgemeinen die Konstruktionszeiten, was sich so erklären läßt dass durch die rekursive Konstruktion der Monome mit höherem Faktor a mehr Prädikate erzeugt werden und mehr Konjunktionen berechnet werden müssen. Ausnahmen lassen

Terme	Vars	A	Ω	Konstruktion	Leerheit 5	Leerheit 10
1·x+1·y=c	14	5192	4983	1,41	> 10 min	> 10 min
2·x+2·y=c	18	16209	20362	25,57	> 10 min	> 10 min
3·x+3·y=c	22	77733	77793	63,67	> 10 min	> 10 min
4·x+4·y=c	22	8538	13281	4,03	> 10 min	> 10 min
5·x+5·y=c				> 10 min		
6·x+6·y=c	26	88308	176204	161,71	> 10 min	> 10 min
10·x+10·y=c	30	74865	208872	214,92	> 10 min	> 10 min
30·x+30·y=c				> 10 min		

Abbildung 33: Konstruktion von Gleichungen mit 3 Monomen in der Basis 3

Terme	Vars	A	Ω	Konstruktion	Leerheit 5	Leerheit 10
1·x+1·y+1·z=c	22	236306	203495	292,32	> 10 min	> 10 min
2·x+2·y+2·z=c	28	21296	30369	28,01	> 10 min	> 10 min
3·x+3·y+3·z=c	34	98813	107407	210,9	> 10 min	> 10 min
4·x+4·y+4·z=c	34	25893	35267	75,65	> 10 min	> 10 min
5·x+5·y+5·z=c				> 10 min		
6·x+6·y+6·z=c	40	55092	100797	557,43	> 10 min	> 10 min

Abbildung 34: Konstruktion von Gleichungen mit 3 Monomen in der Basis 3

Prädikat	Vars	A	Ω	Konstruktion	Leerheit 5
EQ0	1	6	6	0	0
LT0C	1	6	7	0	0
LT0	2	142	265	0,04	0,05
EQ	2	258	265	0,04	0,02
LT	3	728	1334	0,16	> 10 min
ADDC	3	587	680	0,07	1,63
ADD	4	1226	1759	0,1	> 10 min
SUBC	3	307	403	0,07	1,22
SUB	4	1027	1334	0,16	> 10 min

Abbildung 35: Konstruktion von Prädikaten in der Basis 8

Monome	Vars	A	Ω	Konstruktion	Leerheit 5
1·x = y	6	1191	1575	0,18	> 10 min
2·x = y	8	6211	15172	0,87	> 10 min
3·x = y	10	16135	56613	10,66	> 10 min
4·x = y	10	18015	45181	6,12	> 10 min
5·x = y	12	170921	375094	181,94	> 10 min
6·x = y	12	175815	720765	302,59	> 10 min
10·x = y	14			> 10 min	
30·x = y	20			> 10 min	
300·x = y	28			> 10 min	

Abbildung 36: Konstruktion von Monomen in der Basis 8

Terme	Vars	A	Ω	Konstruktion	Leerheit 5
1·x = c	6	2023	2084	0,33	17,11
2·x = c	8	14547	37967	7,45	> 10 min
3·x = c	10	40846	118627	107,6	> 10 min
4·x = c	10	9984	20229	31,21	> 10 min
5·x = c	12			> 10 min	
6·x = c	12			> 10 min	
10·x = c	14			> 10 min	
30·x = c	20			> 10 min	
300·x = c	28			> 10 min	

Abbildung 37: Konstruktion von Gleichungen in der Basis 8

Terme	Vars	A	Ω	Konstruktion	Leerheit 5
1·x + 1·y = c	14	78837	111178	180,98	> 10 min
2·x + 2·y = c	18			> 10 min	
3·x + 3·y = c	22			> 10 min	
4·x + 4·y = c	22			> 10 min	
5·x + 5·y = c				> 10 min	
6·x + 6·y = c	26			> 10 min	

Abbildung 38: Konstruktion von Gleichungen mit 2 Monomen in der Basis 8

Terme	Vars	A	Ω	Konstruktion	Leerheit 5
1·x + 1·y + 1·z = c	22			> 10 min	
2·x + 2·y + 2·z = c	28			> 10 min	
3·x + 3·y + 3·z = c	34			> 10 min	
4·x + 4·y + 4·z = c	34			> 10 min	
5·x + 5·y + 5·z = c				> 10 min	
6·x + 6·y + 6·z = c	40			> 10 min	

Abbildung 39: Konstruktion von Gleichungen mit 3 Monomen in der Basis 8

Alternativen	DFA	DFA	AFA	AFA	AFA	AFA
Terme	Kon	Leer	Kon	Leer 32	Leer 10	Leer 5
1·x=c	0	0	0	0,01	0	0
2·x=c	0	0	0	0,01	0	0
3·x=c	0	0	0	0,04	0,01	0
4·x=c	0	0	0	0,01	0	0
5·x=c	0	0	0	0,06	0,01	0
6·x=c	0	0	0	0,05	0	0
10·x=c	0	0	0	0,06	0,01	0
30·x=c	0	0	0	0,1	0,01	0,01
300·x=c	0	0	0	0,13	0,01	0,01
1·x+1·y=c	0	0	0	0,04	0	0
2·x+2·y=c	0	0	0	0,04	0,01	0
3·x+3·y=c	0	0	0	0,29	0,02	0,01
4·x+4·y=c	0	0	0	0,04	0	0
5·x+5·y=c	0	0	0	0,57	0,02	0,01
6·x+6·y=c	0	0	0	0,11	0,01	0
300·x+300·y=c	0	0	0	0,3	0,04	0,01
1·x+1·y+1·z=c	0	0	0	0,08	0,01	0,01
2·x+2·y+2·z=c	0	0	0	0,09	0,01	0,01
3·x+3·y+3·z=c	0	0	0	1,98	0,09	0,02
4·x+4·y+4·z=c	0	0	0	0,08	0,01	0
5·x+5·y+5·z=c	0	0	0	5,4	0,2	0,02
6·x+6·y+6·z=c	0	0	0	0,22	0,02	0,01

Abbildung 40: Konstruktion von DFAs und AFAs

sich durch kleinere erzeugte BDDs und damit durch kürzere Bearbeitungszeiten erklären.
Gleichungen der Art $a \cdot x = c$ verhalten sich analog zu den Monomen. Werden in den Gleichungen aber mehr Monome verwendet so steigen sowohl die BDD-Größen als auch die Konstruktionszeiten. Es treten bei der Konstruktion dann Time-Outs auf. Bei der Basis 8 treten wesentlich häufiger Time-Outs auf.
Betrachtet man die Zeiten für die Prüfung auf Leerheit so sieht man dass bei 10 Schritten bei praktisch allen Automaten Time-Outs auftreten. Eine Halbierung der Schritte auf 5 verbessert die Ergebnisse der Prüfung nur in wenigen Fällen.
Die Konstruktionszeiten für DFAs und AFAs liegen in allen verwendeten Beispielen unter einer messbaren Schranke. Für DFAs gilt dies auch für die Prüfung der Leerheit. Bei AFAs liegen die Zeiten für diese Prüfung in annähernd allen Fällen unter einer Sekunde.
Vergleicht man die Ergebnisse der Signed-Digit-Automaten mit denen der endlichen und der alternierenden Automaten zeigt sich deutlich dass die Klasse der Signed-Digit-Automaten äußerst ineffizient arbeitet. Von Tests in der Modellprüfung mittels Averest wurde daher abgesehen.

3.4 Ausblick

Für zukünftige Arbeiten wäre es von Interesse die Prüfung auf Leerheit zu verbessern. Dies kann auf mehrere Arten geschehen. Wird die in dieser Arbeit vorgestellte Methode beibehalten so kann daran gedacht werden die durchzuführenden Schritte zu minimieren. Da eine erfüllbare Presburger Formel ein Modell besitzt welches von allen Modellen dieser Formel das kleinste ist, gibt es dür den Automaten ein kürzestes Wort das akzeptiert wird. Läßt sich die Länge dieses Wortes bestimmen, so ist die Anzahl durchzuführender Schritte bekannt.
Da sich die hier vorgestellte Methode als inpraktikabel erwiesen hat, ist es vielleicht erfolgversprechender eine alternative Methode zu wählen. Die Übersetzung des Signed-Digit-Automaten in eine andere Automaten-Klasse oder die Simulation der Transitionsrelation einer anderen Klasse wären zu untersuchende Möglichkeiten.
Die Prüfung der Akzeptanz eines Wortes ist ähnlich problematisch wie die Prüfung der Leerheit, da in beiden Fällen das gleiche Verfahren verwendet wird. Eine Veränderung der Methode der Akzeptanzprüfung könnte darauf untersucht werden, ob kleinere BDDs entstehen und so die Zeiten sinken. Es wären dafür zunächst die BDDs aus A und Ω zu erzeugen indem die Variablenbelegungen durch das Wort vorgenommen werden. Danach wären die verbleibenden BDDs, welche den möglichen Belegungen der freien Variablen entsprechen, auf eine Konsistente Folge der verbliebenen Variablen zu untersuchen. So wären weniger Variablen in eventuell kleineren BDDs zu vertauschen. An der Prüfung auf Leerheit würde diese Änderung wenig verbessern, da nur die Platzhalter der Konstanten eingesetzt werden können, und alle weiteren Variablen wie freie behandelt werden müssen.
Die Konstruktion der Automaten ist unter Verwendung der BDDs zur Repräsentation von Mengen von Signed-Digits auch recht ineffizient. Eine Möglichkeit der Verbesserung besteht darin nach einer Basis zu suchen die zu kleinen BDD-Größen führt. Es ist jedoch nicht gesichert dass eine solche Basis existiert, daher könnte auch nach einer alternativen Darstellung der Signed-Digit-Mengen gesucht werden.

4 Zusammenfassung

In dieser Arbeit wurde die Klasse der Signed-Digit-Automaten vorgestellt, und ihre Fächer-Struktur beschrieben. Es wurden die Automaten zu Prädikaten für Addition und Subtraktion, für Gleichheit und Kleiner definiert. Es wurden Operationen auf den Automaten wie Konjunktion, Disjunktion und die Prüfung auf Leerheit vorgestellt.
Es wurde eine Möglichkeit entwickelt die Transitionen der Automaten, welche aus Mengen von Signed-Digits bestehen, symbolisch durch BDDs zu repräsentieren. Es wurde gezeigt wie die BDDs für die die Transitionen der Prädikate konstruiert werden, und wie die Operationen auf den Automaten unter Verwendung dieser BDDs durchgeführt werden können.
Weiter wurde gezeigt wie ein existenzielles Fragment der Presburger Arithmetik in einen Signed-Digit-Automat übersetzt werden kann. Dazu wird auf die Konstruktion von Automaten für Gleichungen und Ungleichungen und die anschließende Verknüpfung dieser Automaten zurückgegriffen.
Die Prüfung auf Leerheit erlaubt es dann zu entscheiden, ob diese Formel ein Modell besitzt. Die Entscheidung von existenziellen Fragmenten der Presburger Arithmetik ist also mit Signed-Digit-Automaten möglich.
Die vorgestellten Verfahren wurden im Rahmen von Averest implementiert um im praktischen Einsatz eine Bewertung dieses Entscheidungsverfahrens zu erstellen. Eswurden einfache Gleichungen übersetzt und die Größen der BDDs der Transitionen zusammen mit den Zeiten für die Konstruktion und die Prüfung auf Leerheit gemessen. Dabei hat sich gezeigt, dass die Zeit die für die Konstruktion benötigt wird von den Größen der erzeugten BDDs und der Anzahl der durchgeführten Automatenoperationen abhängt.
Die Prüfung auf Leerheit benötigt selbst für die einfachen verwendeten Gleichungen schon lange Zeiten. Häufig treten sogar Time-Outs auf, die verwendeten Algorithmen wurden dann nach 10 Minuten ohne Ergebnis abgebrochen.
Im Vergleich mit endlichen und alternierenden Automaten zeigt sich deutlich dass die Algorithmen der Signed-Digit-Automaten äußerst ineffizient arbeiten und eine Verwendung in der Modellprüfung in der Praxis nicht möglich ist. Es gibt jedoch Ansätze um die Effizienz der Verfahren zu verbessern. Diese näher zu bestimmen und zu untersuchen bleibt jedoch zukünftigen Arbeiten vorbehalten.

Literatur

[1] Randal E. Bryant. *Graph-Based Algorithms for Boolean Function Manipulation.* IEEE Transactions on Computers, Vol C-35, No. 8, August, 1986, Seiten 677 - 691.

[2] Moj$\dot{z}$esz Presburger. *Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige hervortritt.* In F. Leja, editor, Sprawozdanie z I Kongresu Matematyk$\acute{o}$w Slowania$\acute{n}$skich, Warszawa 1929, Seiten 92-101.

[3] D.C. Cooper. *Theorem proving in arithmetic without multiplication.* Machine Intelligence, 7:91-99, 1972

[4] M. Fischer und M.O. Rabin. *Super-exponential complexity of Presburger arithmetic.* In R.M. Karp, editor Complexity of Computation, Volume 7, Seiten 27-41, Providence, RI, 1974, American Mathematical Society (AMS)

[5] D.C. Oppen. *A* $2^{2^{2^{pn}}}$ *upper bound on the complexity of Presburger arithmetic.* Journal of Computer and System Sciences, 16:323-332, 1978

[6] J. R. Büchi. *Weak second-order arithmetic and finite automata.* Zeitschrift Math. Logik und Grundlagen der Mathematik, 6:66-92, 1960

[7] J. R. Büchi. *On a decision method in restricted second order arithmetic.* In Proc. Internat. Congr. Logic, Method und Philos. Sci 1960, Seiten 1-12, Stanford, 1962, Stanford University Press

[8] Pierre Wolper und Bernard Boigelot. *On the Construction of Automata from Linear Arithmetic Constraints* In Tools and Algorithms for the Construction and Analysis of Systems (TACAS). Volume 1785 of LNCS. Berlin, Springer-Verlag, 2000

[9] E. Clarke, O. Grumberg, D. Peled: *Model Checking.* The MIT Press, Cambridge, Massachusetts, London, England, 2000.

[10] K. Schneider. *Verification of Reactive Systems - Formal Methods and Algorithms.* Texts in Theoretical Computer Science (EATCS Series). Springer, 2003.

[11] Tobias Schüle. *Signed Digit Automata.* Universität Kaiserslautern, 2005

[12] Tobias Schüle und Klaus Schneider. *Symbolic Model Checking by Automata Based Set Representation* Universität Karlsruhe, 2002

[13] Tobias Schüle und Klaus Schneider. *Exact Runtime Analysis Using Automata-Based Symbolic Simulation* Universität Kaiserslautern, 2003

[14] *http://www.averest.org*

[15] *http://sourceforge.net/projects/buddy*

[16] *http://vlsi.colorado.edu/∼fabio/CUDD/*

[17] *http://www.princeton.edu/∼chaff/zchaff.html*