ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing Group

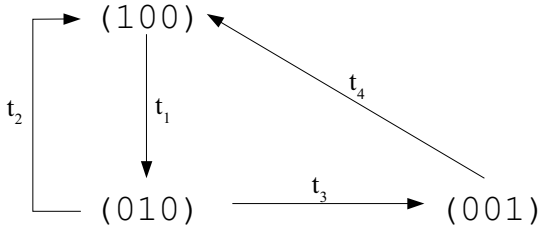
HS 2008 Dr. Kai Lampka / Raphael Eidenbenz / Roland Flury

Exercise 7
Musterlösungen zu Aufgaben 5 und 6

Aufgabe 5: BDD-basierte Komposition von PN

1. Zeichnen Sie das BDD für das gegebene PN.

Zuerst untersuchen wir die möglichen Tokenverteilungen im PN und bauen den Erreichbarkeitsbaum. Die Knoten stellen die Tokenverteilung dar: (x,y,z) steht dafür, dass es im Platz P_1 x Tokens, im Platz P_2 y Tokens und im Platz P_3 z Tokens hat.



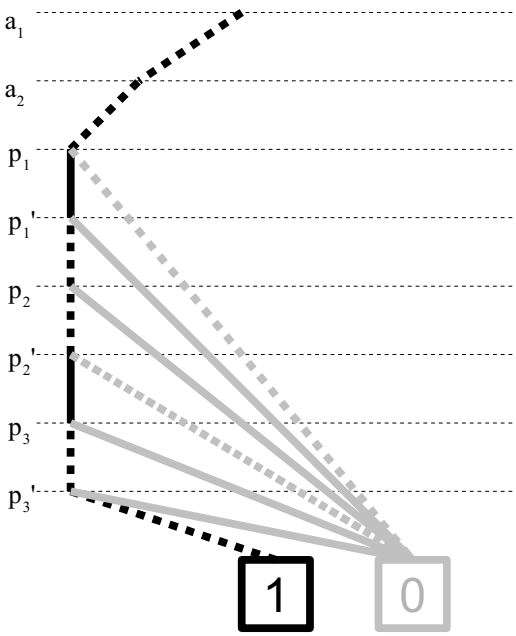
Die Codierung der Transitionsrelation ist eine binäre Darstellung des Erreichbarkeitsbauemes und beschreibt ebenfalls, wie sich die Tokens im PN verschieben. Wir benützen die Variablen a_1 und a_2 um die Transitionen $\{t_1, t_2, t_3, t_4\}$ binär zu codieren.

Die Codierung der Plätze wird uns dadurch erleichtert, als dass wir ein speziell einfaches PN haben, wo in jedem Platz maximal 1 Token ist. D.h. das PN ist *1-bounded*, man sagt auch, das PN sei *safe*. Dank dem können wir die Anzahl Tokens in jedem Platz mit einem Bit angeben (p_1, p_2, p_3). Die zusätzlichen Variablen p_1', p_2', p_3' beschreiben die Tokenverteilung nach dem Feuern der gegebenen Transition. Die Transitionsrelation vom PN kann man mit folgender Tabelle beschreiben, wobei die Tabelle folgendermassen zu lesen ist: Um t_1 zu feuern (codiert mit a_1=a_2=0), muss p_1=1 und p_2=p_3=0 sein. Nach dem Feuern ist die Tokenverteilung p_1'=p_3'=0, und p_2'=1. Dasselbe gilt analog für t_2, t_3 und t_4.

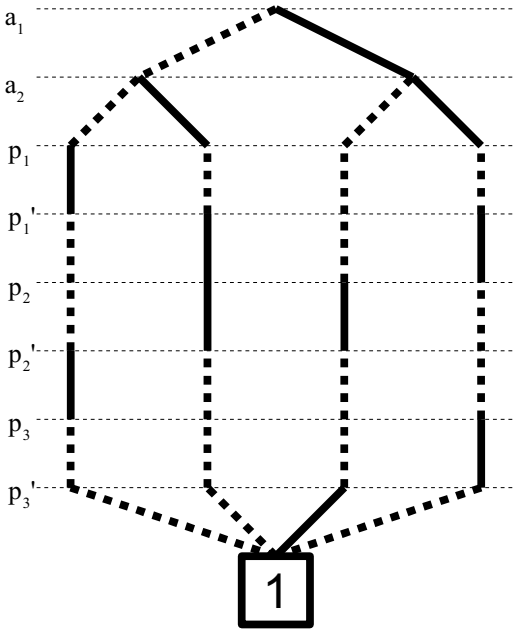
			P_1		P_2		P_3	
	a_1	a_2	p_1	p_1'	p_2	p_2'	p_3	p_3'
t_1	0	0	1	0	0	1	0	0
t_2	0	1	0	1	1	0	0	0
t_3	1	0	0	0	1	0	0	1
t_4	1	1	0	1	0	0	1	0

Das BDD erhalten wir nun von dieser Tabelle indem wir die möglichen Pfade einzeichnen. Dazu zeichnen wir die Transitionen nacheinander ein. Zur Erinnerung: Das BDD beschreibt die boolsche Funktion, die von der obigen Wahrheitstabelle beschrieben wird. Wir zeichnen waagrechte Linien für die Variablen ein (a_1, a_2, p_1, p_1', … p_3'). Ausgehend von jeder Variable beschreiben wir nun was passiert, wenn sie 0, respektive 1 ist. Der Wert 0 (*false*) wird mit einer gestrichenen Linie dargestellt, und der Wert 1 (*true*) mit einer durchgezogenen Linie.

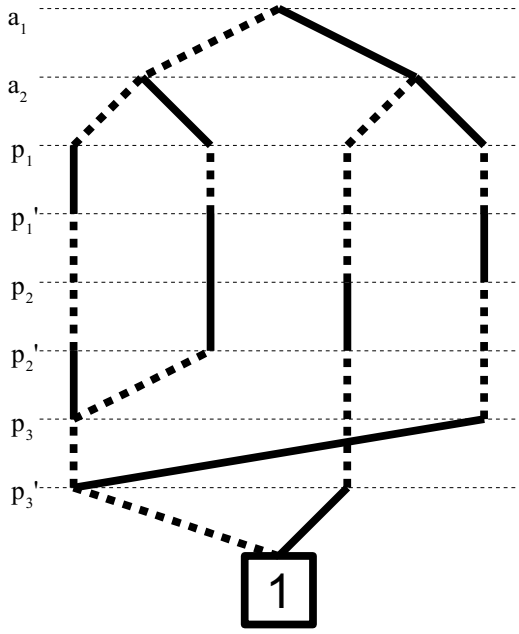
Die Abbildung unten zeigt das BDD für die Transition t_1: a_1 und a_2 sind *false*, p_1 ist *true*, usw. Die letzte Linie endet im *terminalen 1-Knoten*, welcher besagt, dass dieser Pfad von der boolschen Funktion erkannt wird. Der *terminale 0-Knoten* lassen wir normalerweise aus, da diese Information redundant ist (alles was nicht in den terminalen 1-Knoten geht wird von der boolschen Funktion nicht beschrieben.) Z.B. die *false* Kante ausgehend von p_1 würde zum terminalen 0-Knoten gehen, da die Transition t_1 mit $a_1=a_2=0$ nur dann gefeuert werden kann, wenn $p_1=1$.



Zeichnet man alle Transitionen ein, erhält man folgendes BBD.



Ein Vorteil von einem BBD gegenüber der Transitionsrelation in Tabellenform ist, dass wir das BBD vereinfachen können. D.h. Falls ein Teilbaum mehrere male vorhanden ist, können die zwei eingehenden Kanten zusammengeführt werden. In unserem Beispiel kann nur an 2 Orten vereinfacht werden, z.B. bei p_3 hat es zwei mal einen Teilbaum der From <*false*, *false*>, welcher zum terminalen 1-Knoten führt. Das vereinfachte BDD sieht folgendermassen aus:



Anmerkung 1: In unserem Fall konnten wir nur wenig vereinfachen. In grösseren System ist die Wahrscheinlichkeit für equivalente Teilbäume sehr gross und führt zu erheblichen Vereinfachungen, was zu kleineren BDDs führt. Dies wiederum ist wichtig um die Rechenzeit der Operationen auf den BDDs zu verkürzen.

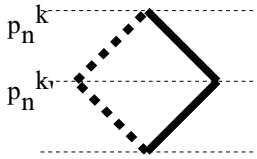
Anmerkung 2: Das Tokengame kann nicht nur im PN, sondern auch auf dem BDD gespielt werden. Wir wissen die initiale Tokenverteilung, e.g. (1,0,0), was p_1=1, p_2=p_3=0 entspricht. Das BDD beschreibt wie die neue Tokenverteilung aussieht, wenn wir eine aktivierte Transition (in diesem Falle t_1 mit a_1=a_2=0) feuern: Wir beginnen immer oben im BDD. Für t_1 haben wir also a_1 ist *false* und nehmen *false* (nach links). a_2 ist ebenfalls *false* und p_1 ist *true*. p_1' ist die Anzahl Tokens in P_1 nach dem Feuern von t_1. Per se wissen wir p_1' nicht, können nun aber dessen Wert aus dem BDD ablesen: p_1'=*false*=0. Das Spiel geht im selben Muster weiter: Wir kennen p_2 und kriegen p_2'....
Versuchen wir auf die Tokenverteilung (1,0,0) eine nicht aktivierte Transition anzuwenden, z.B. t_3, so führt uns das BDD in den (nicht gezeichneten) terminalen 0-Knoten, was aufzeigt, dass für die gegebene Tokenverteilung t_3 nicht gefeuert werden kann.

2. Nehmen Sie zwei der obigen PN und lassen Sie sie parallel (unabhängig voneinander) laufen. Bestimmen Sie das BDD welches die Transitionsrelation beider PNs beschreibt.

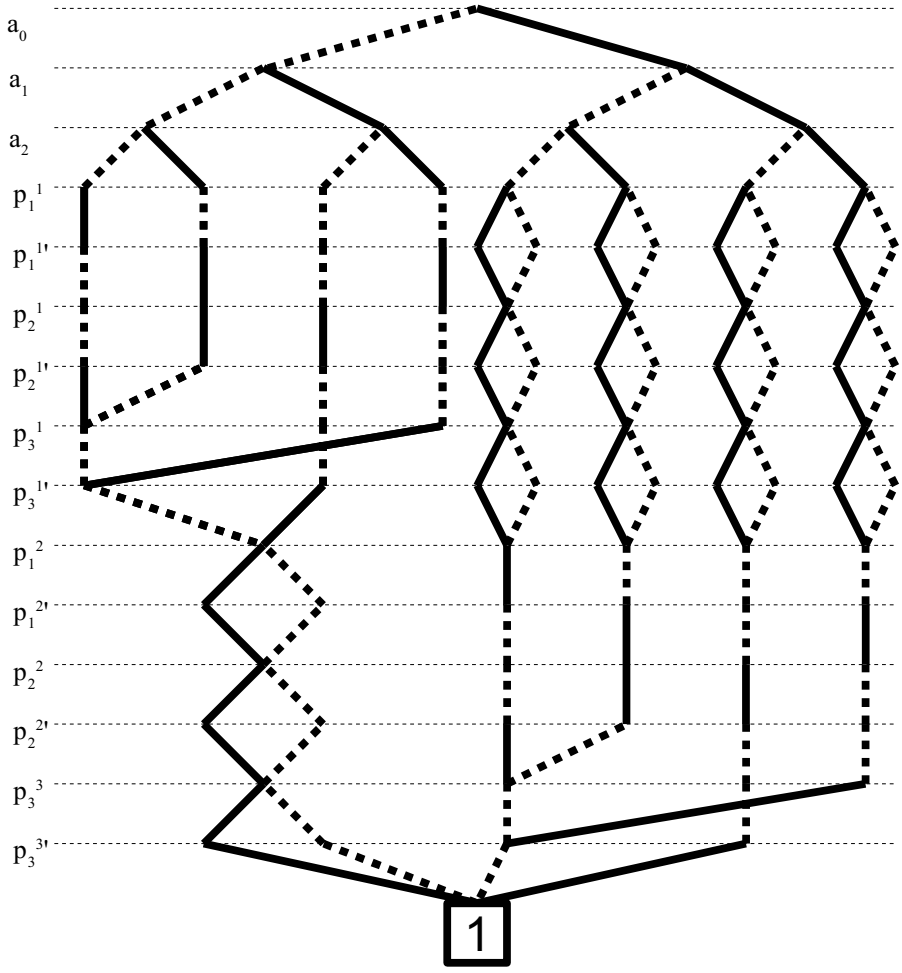
Wir könnten wiederum vorgehen wie in der ersten Aufgabe, und die Transitionsrelation als Tabelle angeben. Um die verschiedenen Variablen der zwei PN zu unterscheiden führen wir Superskripte ein – ^{1} für das erste PN und ^{2} für das zweite PN. Die Transitionen codieren wir mit a_0, a_1, a_2, wobei a_0 0 (1) ist falls es sich um eine Transition im ersten (zweiten) PN handelt. a_1 und a_2 codieren die entsprechende Transition wie in Aufgabe 1.

					P_1^1		P_2^1		P_3^1		P_1^2		P_2^2		P_3^2	
		a_0	a_1	a_2	p_1^1	$p_1^{1'}$	p_2^1	$p_2^{1'}$	p_3^1	$p_3^{1'}$	p_1^2	$p_1^{2'}$	p_2^2	$p_2^{2'}$	p_3^2	$p_3^{2'}$
	t_1^1	0	0	0	1	0	0	1	0	0						
PN 1	t_2^1	0	0	1	0	1	1	0	0	0				I^2		
	t_3^1	0	1	0	0	0	1	0	0	1						
	t_4^1	0	1	1	0	1	0	0	1	0						
	t_1^2	1	0	0							1	0	0	1	0	0
PN 2	t_2^2	1	0	1				I^1			0	1	1	0	0	0
	t_3^2	1	1	0							0	0	1	0	0	1
	t_4^2	1	1	1							0	1	0	0	1	0

Wenn eine Transition in PN1 feuert, dann bleibt die Tokenverteilung in PN2 konstant. Dies ist dargestellt mit der Identität I^2, welche die Tokenverteilung in PN2 nicht verändert. Dasselbe gilt für PN2 und I^1. Für alle Einträge in der Identität gilt, dass $p_n^k = p_n^{k'}$, was als BDD als Route dargestellt wird:



Das neue BDD sieht folgendermassen aus:



Wiederum könnten wir auf diesem BDD das Tokengame spielen und aus einer gegebenen Tokenverteilung (p_1^1, p_2^1, p_3^1, p_1^2, p_2^2, p_3^2) die neue Tokenverteilung ($p_1^{1'}$, $p_2^{1'}$, $p_3^{1'}$, $p_1^{2'}$, $p_2^{2'}$, $p_3^{2'}$) ableiten.

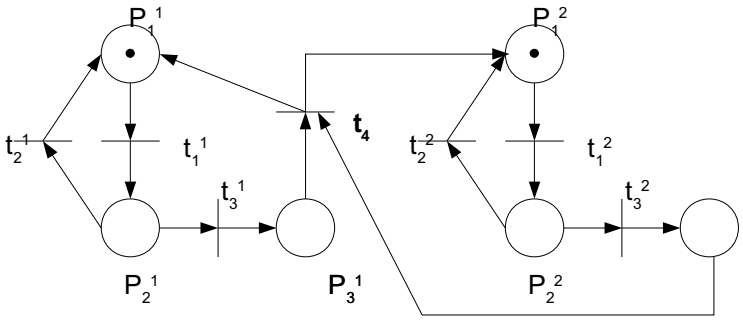
Problematisch an der Identität ist, dass sie nicht eine korrekte Tokenverteilung erzwingt, wie das im restlichen Teil des BDDs der Fall ist. D.h. das BDD würde die Transition t_1 auf die Tokenverteilung (1, 0, 0, 1, 1, 1) im terminalen 1-Zustand beenden, obwohl das zweite PN nie die Tokenverteilung (1, 1, 1) erreichen kann. Das heisst, das konstruierte BDD akzeptiert eine *Übermenge* der boolschen Funktion die es darstellen sollte.

Dies ist aus zwei Gründen kein Problem:

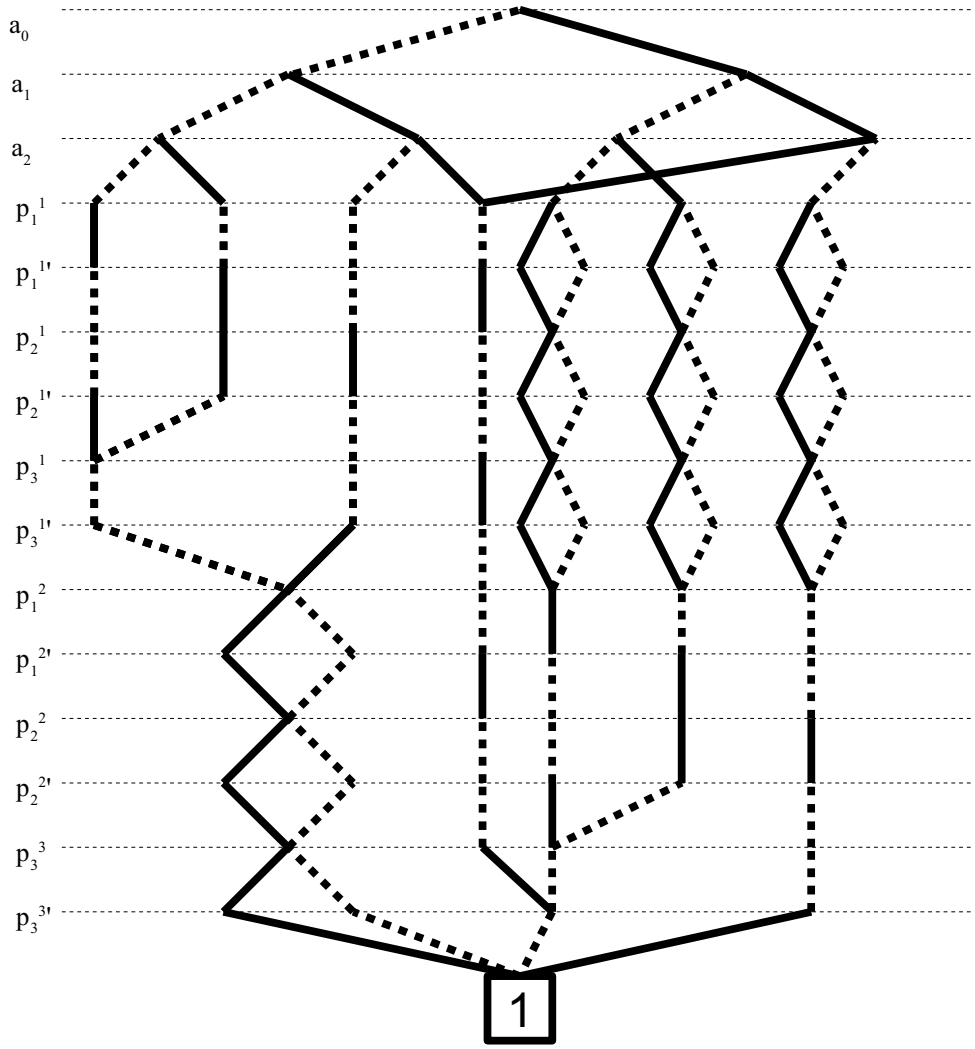
1) Das Tokengame kann immer noch korrekt gespielt werden. Wird dem BDD eine *korrekte* Tokenverteilung als Eingabe gegeben, gibt es auch wieder eine korrekte Tokenverteilung als Antwort.
2) Die Erreichbarkeit aller Tokenverteilung (z.B. bestimmbar mit dem Algorithmus auf Folie 3/88 ist immer noch gegeben.)

3. Wie müssen Sie nun ihren Ansatz ändern, wenn Transition t_4^1 und t_4^2 immer synchron miteinander ausgeführt werden müssen?

Um die zwei Transitionen zu synchronisieren werden sie zu einer einzelnen zusammengefasst. Das PN sieht dann folgendermassen aus:



Für das BDD hat das zur Folge, dass wir nicht mehr t41 und t42 haben, sondern nur noch die Transition t4, welche die zwei Ereignisse synchronisiert:



Aufgabe 6: Zero-suppressed BDDs

Es sind nur diejenigen Fälle zu berücksichtigen, in denen eine Variable übersprungen wurde (was einem zero-suppressed Knoten entspricht). Somit sehen die Zeilen 10 bis 17 folgendermassen aus:

(10) ELSE if var(n) < var(m) THEN

(11) v := var(n)

(12) e := Apply(op, else(n), m)

(13) t := Apply(op, then(n), **0-node**)

(14) ELSE

(15) v := var(m)

(16) e := Apply(op, n, else(m))

(17) t := Apply(op, **0-node**, then(n))

Die Operatoren für ZBDDs müssen 0-erhaltend sein, also 0 op 0 = 0 (ansonsten müsste aus zwei nicht existierenden Knoten einen neuen Knoten generiert werden!)

Weitere infos to BDDs and ZDDs:

Source: *An Introduction to Zero-Suppressed Binary Decision Diagrams* by Alan Mishchenko, *www.eecs.berkeley.edu/~alanmi/publications/2001/tech01_zdd.pdf*

Fifteen years ago, Binary Decision Diagrams (BDDs) [4] and their variations entered the scene of computer design. Since that time they have been often used in research software and industrial CAD tools. The role of BDDs is two-fold. They are used as (1) a memory-efficient storage and convenient processing media for Boolean or multivalued functions and (2) a representation facilitating the analysis of data leading to new implicit algorithms, which tend to be more efficient than the classical ones [9].
One of the reasons why decision diagrams, and in particular BDDs, became useful for the CAD tool developers, is that they provide canonical representation of discrete objects. Canonical means that under certain condition for every object there is only one representation of this kind. This property is extremely important for verification because in order to prove the identity of two objects it is sufficient to build their canonical representations and show that these representations are identical. This property is also important it synthesis. For example, canonical representations are useful as signatures when storing objects in hash tables.
The experience of using BDDs in numerous applications shows that they are not a panacea for all types of problems. In some cases, due to the specific properties of the discrete data arising in particular settings, the BDDs grow large making processing inefficient or impossible. In particular, this situation occurs when the applications work with sparse sets represented by characteristic functions [5].
A set is sparse if the number of elements in it is much smaller than the total number of elements that may appear in the set. Cube covers are an example of sparse sets, because a typical cube contains only a few literals, out of all possible literals that may appear in the cube. The maximum number of literals is reached when a cube is a minterm containing each variable as either the negative or the positive literal. In the case of minterms, the sparseness of the set is ½, because each minterm contains exactly one half of all possible literals that may appear in the cubes.
The problem of prohibitively large BDD size of the sparse set representation can be remedied by introducing a different brand of decision diagrams, called Zero suppressed binary Decision Diagrams (ZDDs) [21]. These diagrams are similar to BDDs with one of the underlying principles modified. The latter explains the improved efficiency of ZDDs when handling sparse sets and a number of other semantic differences between the two types of diagrams.
While BDDs are better for the representation of functions, ZDDs are better for the representation of covers. Additionally, there are efficient procedures to perform conversions between them. Taken together, BDDs and ZDDs provide a powerful framework to solve problems in logic synthesis, such as two-level sum-of-product (SOP) minimization [8], three-level minimization, factorization [27][35], and decomposition [18].
The use of ZDDs is not limited to logic synthesis. They have been used, independently of BDDs, in a number of applications, ranging from the graph-theory problems to handling polynomials and regular expressions. (See Section 8 for what is intended to be a complete list of ZDD applications published to date.)

Both BDDs and ZDDs can be seen as decision trees, simplified using two reduction rules that guarantee the canonicity of the resulting representation. The second reduction rule (merging of isomorphic subgraphs) holds for both BDDs and ZDDs; however, they differ in the first reduction rule (node elimination).
For BDDs, the node is removed from the decision tree if both its edges point to the same node. For ZDDs, the node is removed if its positive edge (then-edge) points to the terminal node 0. This variation in the rule, as mentioned before, explains the improved efficiency of ZDDs when handling sparse sets and the semantic differences between the two types of diagrams.
One way of understanding the principles of ZDDs is to compare them with BDDs for simple illustrative functions while keeping in mind their main difference.