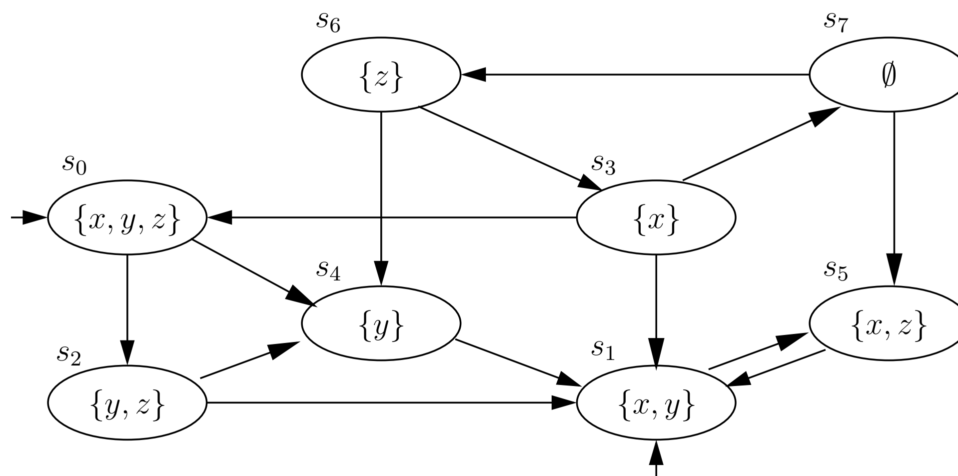
Übungen zu Grundlagen der Softwarezuverlässigkeit

Abgabe bis zum 27.01.05

Aufgabe 6.1 Model Checking



Betrachten Sie die CTL-Formel $\varphi = \neg y \, \mathbf{AU} \, \neg x$.

(a) Wandeln Sie φ in eine äquivalente Formel φ' um, die keine Temporaloperatoren außer **EG** und **EU** enthält.

(b) Wenden Sie sodann den Fixpunkt-Model-Checking-Algorithmus aus der Vorlesung Schritt für Schritt auf φ' und die oben dargestellte Kripke-Struktur an. (Führen Sie zunächst die Fixpunktberechnung für die innerste Teilformel durch und arbeiten Sie sich nach "außen" vor.) In welchen Zuständen gilt φ?

(c) Betrachten Sie nun die Formel $\psi = \mathbf{AFAG} \, x$ wieder auf obiger Kripke-Struktur.

i) Überführen Sie ψ wieder in eine äquivalente Formel ψ', welche außer **EG** und **EU** keine Temporaloperatoren verwendet.

ii) Wenden Sie nun den Algorithmus von Tarjan zur Bestimmung der starken Zusammenhangskomponenten an:

Sie finden den Algorithmus in Pseudo-Code am Ende des Übungsblattes. Nehmen Sie an, die Knoten $s_0, \ldots, s_7$ sind in genau dieser Reihenfolge in dem Array `G` abgespeichert und für jeden Knoten s_i sind seine Nachfolger $s_{i_1}, \ldots, s_{i_k}$ ebenfalls aufsteigend bzgl. des Index i_l in dem jeweiligen Array `G[i].successor` abgespeichert.

Für s_0 ist somit `G[0].successor = { 2, 4 }` mit `G[0].n_successors = 2`.

Geben Sie dann die Werte `G[i].dfs_index` und `G[i].lowlink` für alle Knoten nach Terminierung des Algorithmus an.

iii) Führen Sie nun den Model-Checking-Algorithmus unter Verwendung der SCCs für ψ' durch.

Aufgabe 6.2 BDDs - Relationales Produkt

Aus der Vorlesung wissen Sie, dass für die CTL-Model-Checking-Algorithmen BDDs zur Repräsentation der verschiedenen Zustandsmengen und -relationen verwendet werden. Insbesondere ist die symbolische Berechnung der Vorgängerzustände einer Menge bezüglich der Übergangsrelation ein entscheidender Bestandteil.

In dieser Aufgabe sollen Sie eine rekursive Funktion `BDD relProd( BDD f, BDD g )` schreiben, welche als Parameter (Referenzen auf - man nehme an, es handle sich um Java) BDD-Knoten `f,g` erhält. Dabei soll angenommen werden, dass `f` eine Menge F von Tupeln über den Variablen $x_1, \ldots, x_k, y_1, \ldots, y_n$ repräsentiert. Entsprechendes gelte für `g` und die Menge G über den Variablen $y_1, \ldots, y_n$. `relProd( f, g )` soll dann einen BDD-Knoten `r` (bzw. eine Referenz darauf) zurückgeben, welcher die Menge

$$\{(x_1, \ldots, x_k) \mid \exists (y_1, \ldots, y_n) \in G : (x_1, \ldots, x_k, y_1, \ldots, y_n) \in F\}$$

repräsentiert, also die Menge der Vorgänger von G bezüglich F. Orientieren Sie sich hierfür an den aus der Vorlesungen bekannten rekursiven Algorithmen zur Konstruktion von BDDs.

Folgende Hilfsfunktionen stehen Ihnen zur Verfügung.

- `BDD ite( BDDVar z, BDD f, BDD g )` erhält als Parameter eine BDD-Variable `z` und BDD-Knoten `f` und `g`. Die Rückgabe ist ein BDD-Knoten `r`, welcher die Formel $(z \wedge f) \vee (\neg z \wedge g)$, bzw. deren erfüllende Belegungen repräsentiert.
- Entsprechend liefert `BDD or( BDD f, BDD g )` einen BDD für $f \vee g$.
- `BDDVar var( BDD f )` liefert für einen BDD-Knoten `f` die zugehörige Variable.
- `boolean exists( BDD f, BDD g )` schaut in einem Cache nach, ob bereits `relProd(f,g)` berechnet wurde und gibt ggf. den entsprechenden BDD-Knoten zurück. Existiert kein solcher Eintrag, so wird `Null` zurückgegeben.
- Mit `void insert( BDD f, BDD g, BDD r )` kann der BDD-Knoten `r` in das Cache an der "Stelle" `(f,g)` eingetragen werden.
- Für zwei Variablen `x,y` liefert `BDDVar min( BDDVar x, BDDVar y )` die bzgl. der festen Variablenordnung kleinere Variable zurück.

  "Kleiner" soll hierbei bedeuten, dass diese Variable zuerst verarbeitet werden muss.
- Mit `FALSE` und `TRUE` sind Referenzen auf die beiden `false` bzw. `true` repräsentierenden BDD-Knoten gegeben.

  `var(FALSE)` bzw. `var(TRUE)` kommen dabei in der Variablenordnung immer an letzter Stelle.
- Ist `f` ein BDD-Knoten, so können Sie mittels `f(0)` bzw. `f(1)` auf seine beiden Nachfolgerknoten zugreifen, d.h.:

  Gilt `z == var(f)`, dann repräsentiert `f(i)` für $i \in \{0, 1\}$ die Formel $f\mid_{z=i}$.
- Für zwei BDD-Knoten `f,g` wertet sich `f == g` zu `true` aus, falls es sich um ein und denselben BDD-Knoten handelt.

Aufgabe 6.3 BDDs

Man betrachte für $n > 0$ folgende aussagenlogische Formel

$$f_n \equiv \bigwedge_{i=0}^{2^n-1} \vec{m} = i \wedge g_{i,n}$$

mit

$$g_{i,n} \equiv \bigvee_{k=0}^{2^n-1} (a_k = b_{(k+i) \bmod 2^n})$$

über den Variablen $m_0, \ldots, m_{n-1}, a_0, \ldots, a_{2^n-1}, b_0, \ldots, b_{2^n-1}$. $\vec{m} = i$ steht dann stellvertretend dafür, dass $m_0 m_1 \ldots m_{n-1}$ gerade die Binärdarstellung der Zahl i ist. Der Wert von $\vec{m}$ selektiert also genau eine der 2^n Funktionen $g_{i,n}$.

(a) Zeichnen Sie für $n = 1$ jeweils den zugehörigen BDD für die Variablenordnungen $m_0 < a_0 < b_0 < a_1 < b_1$ und $a_0 < a_1 < b_0 < b_1 < m_0$.

(b) Zeigen Sie, dass unabhängig von der gewählten Variablenordnung ein BDD für f_n mindestens $2^{2^{n-1}}$ viele Knoten benötigt, d.h., jeder solche BDD ist exponentiell in der Länge von f_n.

Gehen Sie hierzu wie folgt vor:

- Wählen Sie eine beliebige, aber feste Variablenordnung und betrachten Sie die bzgl. der gewählten Ordnung ersten 2^n a- bzw. b-Variablen. Seien dies $A = \{a_{k_1}, \ldots, a_{k_p}\}$ und $B = \{b_{l_1}, \ldots, b_{l_q}\}$ mit $p + q = 2^n$.

  Es sei

  $$S = \{(l_u - k_w) \bmod 2^n \in \{0, \ldots, 2^n - 1\} \mid 1 \leq w \leq p, 1 \leq u \leq q\}$$

  die Menge der *Shifts*, wobei l_u, k_w die Indizes aus B bzw. A sind.

  Zeigen Sie: Für $s \in S$ vergleicht $g_{s,n}$ mindestens ein Variablenpaar aus $A \times B$.

- Wie viele Elemente kann S maximal besitzen? Zeigen Sie dann, dass es ein $s^* \in S$ gibt, so dass $g_{s^*,n}$ maximal 2^{n-2} Variablen*paare* aus $A \times B$ vergleicht.

- Warum müssen dann nach Lesen der Variablen aus $A \cup B$ für $\vec{m} = s^*$ mindestens $2^{2^{n-1}}$ verschiedene Zustände in dem BDD erreicht worden sein?

```
struct Node_s {
   int    n_successors;              // Anzahl der Nachfolger
   int    successor[n_successors];   // Knotenindices der Nachfolger
   bool   visited;                   // wird von dfsSCC benutzt
   bool   on_stack;                  // - " -
   int    dfs_index;                 // - " -
   int    lowlink;                   // - " -
};

class dfsSCC {
protected:
   int          dfs_index;
   stack< int > SCC;          // stack fuer die Knotenindizes

   void recursion( Node Graph[], int u ) {
      ++dfs_index;
      G[u].visited   = true;
      G[u].dfs_index = dfs_index;
      G[u].lowlink   = dfs_index;
      SCC.push( u );
      G[u].on_stack  = true;

      for( int i = 0; i < G[u].n_successors; ++i ) {
         int w = G[u].successor[i];
         if( !G[w].visited ) {
            recursion( G, w );
            G[u].lowlink = min( G[u].lowlink, G[w].lowlink );
         }
         else if( G[w].dfs_index < G[u].dfs_index && G[w].on_stack )
            G[u].lowlink = min( G[u].lowlink, G[w].dfs_index );
      } // end for

      if( G[u].lowlink == G[u].dfs_index ) {
         cout << "SCC gefunden: ";
         do {
            int v = SCC.top();
            SCC.pop();
            G[v].on_stack = false;
            cout << v << " "; // Ausgabe der Knoten der SCC
         } while( u != v );
      } // end if
   } // end recursion
public:
   //Graph G ist ein Array vom Typ Node_s mit Knoten 0 ... n_nodes - 1
   void findSCCs( Node_s G[], int n_nodes ) {
      dfs_index = 0;
      SCC.clear(); // der Stack SCC ist leer

      for( int i = 0; i < n_nodes; ++i ) {
         G[i].visited  = false;
         G[i].on_stack = false;
      }

      for( int i = 0; i < n_nodes; ++i )
         if( !G[i].visited ) recursion( G, i );
   }
};
```