

Übungen zu Model Checking

Besprechung am 30.06.05

Aufgabe 5.1 BDDs

Betrachten Sie folgendes paralleles Programm:

$$P_1 \equiv \text{while true do } \begin{array}{l} x := x + 1 \text{ mod } 3; \end{array} \parallel P_2 \equiv \text{while true do } \begin{array}{l} y := y + 1 \text{ mod } 2; \end{array}$$

Die beiden Prozesse P_1 und P_2 definieren die Kantenrelation einer Kripkestruktur, welche als Zustandsmenge $S = \{(x, y) \in \{0, 1, 2\} \times \{0, 1\}\}$ die Menge aller möglichen Variablenwerte von x und y hat.

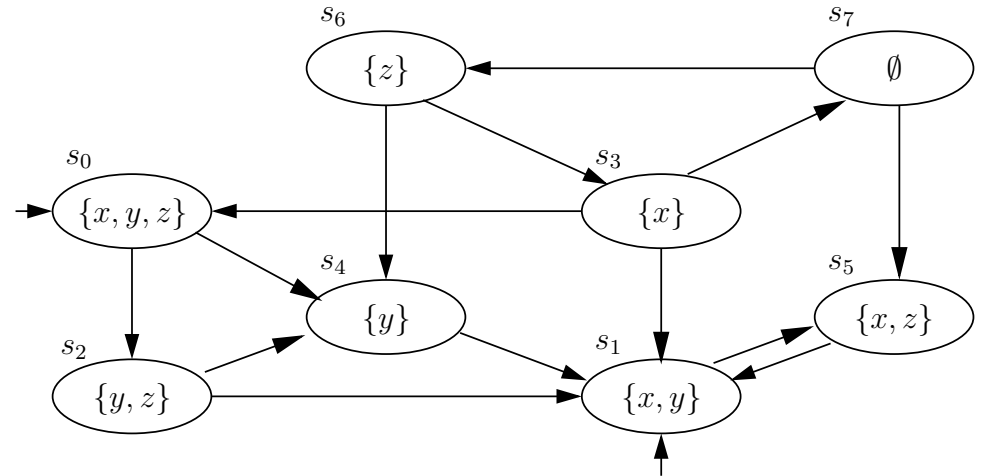
- (a) Konstruieren Sie zunächst für jeden der beiden Prozesse P_i jeweils einen BDD π_i , welcher die von P_i beschriebene Speichertransformation beschreibt. Stellen Sie hierfür die Programmvariablen x, y geeignet mittels boolescher Variablen x_i, y_j dar.

Verwenden Sie wie üblich die Variablen x'_i und y'_j , um die Werte der Variablen x_i bzw. y_j nach Ausführung der entsprechenden Anweisung zu beschreiben.

Wählen Sie schließlich eine Variablenordnung, welche zu möglichst kleinen BDDs führt.

- (b) Konstruieren Sie aus den beiden BDDs π_1 und π_2 der letzten Teilaufgabe einen BDD δ , welcher die Kantenrelation der Kripkestruktur beschreibt, wobei in jedem Zeitschritt *genau einer* der Prozesse P_1 und P_2 ausgeführt wird, während sich der Zustand des anderen Prozesses *nicht* ändern soll.
- (c) Geben Sie den BDD σ an, welcher die Speicherbelegungen $\{(x', y') \in \{(0, 1), (2, 0)\}\}$ repräsentiert.
- (d) Konstruieren Sie aus den BDDs δ und σ den BDD zu der Formel $\exists(x', y') : \delta \wedge \sigma$.

Hinweis: Sie müssen keine Komplementkanten verwenden.



Betrachten Sie die CTL-Formel $\psi = \mathbf{AFAG} x$ auf obiger Kripke-Struktur.

- (a) Überführen Sie ψ in eine äquivalente Formel ψ' , welche außer **EG** und **EU** keine Temporaloperatoren verwendet.
- (b) Führen Sie nun das CTL-Modelchecking mit Hilfe des Algorithmus von Tarjan durch. Sie finden den Algorithmus in Pseudo-Code am Ende des Übungsblattes. Nehmen Sie an, die Knoten s_0, \dots, s_7 sind in genau dieser Reihenfolge in dem Array \mathbf{G} abgespeichert und für jeden Knoten s_i sind seine Nachfolger s_{i_1}, \dots, s_{i_k} ebenfalls aufsteigend bzgl. des Index i_i in dem jeweiligen Array $\mathbf{G}[i].\text{successor}$ abgespeichert.

Für s_0 ist somit $\mathbf{G}[0].\text{successor} = (2, 4)$ mit $\mathbf{G}[0].\text{n_successors} = 2$.

Geben Sie dann die Werte $\mathbf{G}[i].\text{dfs_index}$ und $\mathbf{G}[i].\text{lowlink}$ für alle Knoten nach Terminierung des Algorithmus an.

```

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 );
};

```