MRMC as a model checker for CTMCs

Quantitative Logics

May 22, 2013

David N. Jansen

MRMC

- Markov Reward Model Checker
- model checker for:
  - Markov chains
  - Markov reward models
- research tool:
  thorough mathematics, no fancy user interface
- authors: Joost-Pieter Katoen (initiator), Ivan Zapreev, Maneesh Khattri (most code), David N. Jansen (bisimulation), ...

Hubble’s gyroscopes as a MRMC input file

Transitions

```
STATES 7
TRANSITIONS 8
1 7 0.1
1 6 2
2 1 0.2
2 6 2
3 2 0.3
4 3 0.4
5 4 0.5
6 5 0.6
```

State labels

```
#DECLARATION
working crash
#END
3 working
4 working
5 working
6 working
7 crash
```

```
dnjansen% mrmc ctmc hubble.tra hubble.lab
 ------------------------------------------------------------------
|                    Markov Reward Model Checker                   |
|                         MRMC Version 1.5                          |
|               Copyright (C) RWTH Aachen, 2006-2011.               |
|         Copyright (C) The University of Twente, 2004-2008.        |
|                             Authors:                              |
|      Ivan S. Zapreev (2004-2011), David N. Jansen (since 2007),   |
|         E. Moritz Hahn (2007-2010), Falko Dulat (2009-2010),      |
|         Christina Jansen (2007-2008), Sven Johr (2006-2007),      |
|          Tim Kemna (2005-2006), Maneesh Khattri (2004-2005)       |
|           MRMC is distributed under the GPL conditions            |
|           (GPL stands for GNU General Public License)             |
|          The product comes with ABSOLUTELY NO WARRANTY.           |
|  This is a free software, and you are welcome to redistribute it. |
 ------------------------------------------------------------------
 Logic          = CSL
Loading the 'hubble.tra' file, please wait.
States=7, Transitions=8
Loading the 'hubble.lab' file, please wait.
The Occupied Space is ??? Bytes.
Type 'help' to get help.
>>P{<0.01} [tt U[0,15] crash]
```

code walkthrough

- Fox–Glynn algorithm (parts)
- bounded until formulas $U^{[0,\mathit{supi}]}$
- bounded until formulas $U^{[\mathit{subi},\mathit{supi}]}$

Fox–Glynn algorithm

```
typedef struct FoxGlynn {
        int left;
        int right;
        double total_weight;
        double weights[1];          /* struct hack: actually an array of
                                       doubles with (right–left+1) elements*/
} FoxGlynn;
```

```
FoxGlynn * fox_glynn(const double lambda)
{
        const int m = (int) floor(lambda);
        int j, t;
        FoxGlynn * pFG = finder(m, lambda); /* Find lower and upper bound,
                                               set middle element */
        for ( j = m – pFG->left ; j > 0 ; j–– )
                pFG->weights[j–1] = (j+pFG->left) / lambda * pFG->weights[j];
        t = pFG->right – pFG->left;
        for ( j = m – pFG->left ; j < t ; j++ )
                pFG->weights[j+1] = lambda / (j+1 + pFG->left) * pFG->weights[j];
        pFG->total_weight = 0.0;
        j = 0;
        /* t was set above */
        while( j < t )
                if ( pFG->weights[j] <= pFG->weights[t] ) {
                        pFG->total_weight += pFG->weights[j];         j++;
                } else {
                        pFG->total_weight += pFG->weights[t];         t––;
                }
        pFG->total_weight += pFG->weights[j];
        return pFG;
}
```

bounded until

phi U$^{[0,\mathit{supi}]}$ psi

```
static double * bounded_until_universal(
                        bitset *good_phi_states,        /* phi states that can reach psi */
                        const bitset *psi,              /* psi states */
                        double supi                     /* supremum of the interval */
)
{
        int i;
        const int size = get_state_space_size();
        double * reach = (double *) calloc((size_t) size, sizeof(double)),
                * result;
        i = state_index_NONE;
        while ( (i = get_idx_next_non_zero(psi, i)) != state_index_NONE )
                reach[i] = 1.0;
        result = uniformization(good_phi_states, reach, supi);
        free(reach);
        return result;
}
```

```
static double * uniformization(bitset *good_phi_states, double *reach, double supi)
{
        const sparse *state_space = get_state_space();
        const int size = get_state_space_size();
        double exitrate = 0.0, current_fg = 0.0;
        int i, j, non_absorbing=0;
        double * diag = (double *) calloc((size_t) size, sizeof(double));
        double * res = (double *) calloc((size_t) size, sizeof(double));
        double * result = (double *) calloc((size_t) size, sizeof(double));
        /* make relevant states absorbing: */
        sparse * abs_local = ab_state_space(state_space, good_phi_states, &exitrate, diag);
        vec_state_t * valid_rows;
        FoxGlynn * pFG;

        valid_rows = count_set(n_absorbing);
        memcpy(res, reach, sizeof(double)*size );
        memcpy(result, reach, sizeof(double)*size );
        if( (pFG = fox_glynn(exitrate * supi)) != NULL ) {
                Do the calculations...
        }
        Free all temporary data...
        return result;
}
```

The calculations:

```
sub_mtx_diagonal(abs_local, diag);
mult_mtx_const(abs_local, 1 / exitrate);
add_mtx_cons_diagonal(abs_local, 1.0);
for( i=1; i < pFG->left; i++ ) {
        multiply_mtx_cer_MV(abs_local, reach, res, valid_rows);
        swap(reach, res);
}
if( pFG->left == 0 ) {
        Corrections for lower bound == 0 ...
}
for( ; i <= pFG->right; i++ ) {
        current_fg = pFG->weights[i – pFG->left];
        multiply_mtx_cer_MV(abs_local, reach, res, valid_rows);
        int * iterator = valid_rows->el;
        for ( j=0 ; j < valid_rows->count ; j++, iterator++ )
                result[*iterator] += current_fg * res[*iterator];
        swap(reach, res);
}
int * iterator = valid_rows->el;
for ( j = 0 ; j < valid_rows->count ; j++, iterator++ )
        result[*iterator] /= pFG->total_weight;
```

bounded until

phi $U^{[subi,supi]}$ psi

```
double * interval_until(const bitset *phi, const bitset *psi, double subi, double supi) {
        const int size = bitset_size(phi);
        int i;
        const sparse * state_space = get_state_space();
        double * reach = (double *) calloc((size_t) size, sizeof(double));
        double * result1 = NULL, * result2 = NULL;
        bitset *good_phi_states = get_good_phi_states( phi, psi, state_space);
        bitset *phi_and_psi;
        i = state_index_NONE;
        while ( (i = get_idx_next_non_zero(psi, i)) != state_index_NONE )
                reach[i] = 1.0;
        result2 = uniformization(good_phi_states, reach, supi – subi);
        free(reach);
        for( i = 0 ; i < size ; i++ )
                if( ! get_bit_val(phi, i) )
                        result2[i] = 0.0;
        phi_and_psi = and(phi, psi);
        or_result(phi_and_psi, good_phi_states);
        free_bitset(phi_and_psi);
        result1 = uniformization(good_phi_states, result2, subi);
        free_bitset(good_phi_states); free(result2);
        return result1;
}
```