Verify algorithms' properties with the MSR(C) prototype

Alessandro Bulzacchi

Computational Models and Languages

Year 2011-2012

Introduction:

- MSR(C) is a tool to model-check infinite state systems, extending Petri Nets ideas
- In many practical cases verification problems can be reduced to problems related to Petri Nets.
- Reachability procedures can be used to verify the original property on the resulting Petri Net.

Constraint Multiset Rewriting: 1/4

- Petri Nets can be viewed as multiset rewriting systems defined over a finite alphabet P.
- The symbols in P represent the set of places of a Petri Net.
- Petri Net transitions can be modeled as multiset rewriting rules over symbols in P.

Constraint Multiset Rewriting: 2/4

- Given P = { **null, use, idle, busy** }, we can view a multiset M = **null** | **null** | **idle** as a Petri Net marking with two tokens in place **null** and one in place **idle**.
- The multiset rewriting rule

$$\textbf{think} \mid \textbf{idle} \longrightarrow \textbf{use} \mid \textbf{busy}$$

can be viewed as Petri Net transition.

Constraint Multiset Rewriting: 3/4

- If we annotate rules with constraints, we achieve a clear separation between process structure and declarations of data relations.
- In a rule like:

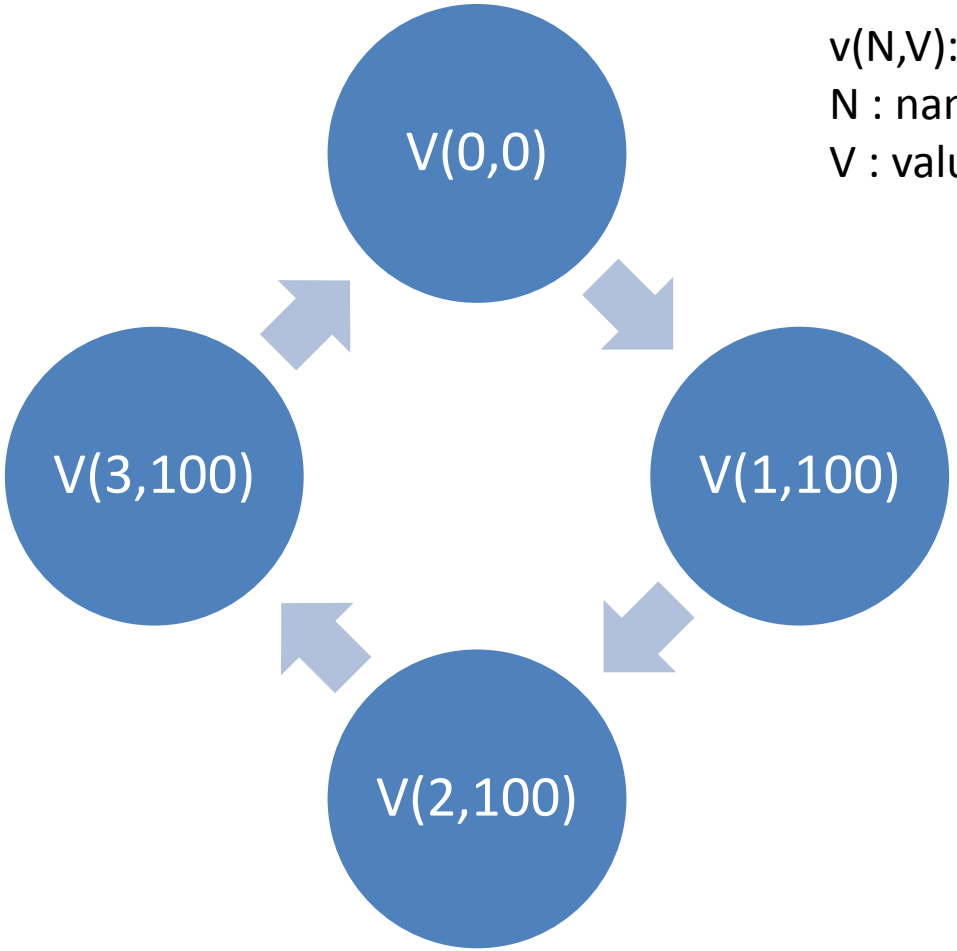
$$\mathbf{think} \mid \mathbf{count}(x) \longrightarrow \mathbf{wait}(y) \mid \mathbf{count}(x0) : x = y \wedge x' = x + 1$$

- The variable x, y, and x' are universally quantified, and $x = y \wedge$, ... is a constraint defined over them.

Constraint Multiset Rewriting: 4/4

- The logic obtained by combining multisets and constraints can be used to reason about system properties.
- Specially, several correctness properties can be verified by searching for error traces following the backward reachability search scheme.

Case-studies: Self-organizing networks: 1/5



v(N,V):
N : name of the node
V : value of the node

Case-studies: Self-organizing networks: 2/5

- *Initial State*

Init $\longrightarrow$ v(0,N) + link(0,1) + v(1,M) + link(1,2) + v(2,M) + link(2,3) + v(3,M) + link(3,0)

- *Rule of Behavior*

v(I,N) + link(I,I2) + v(I2,M) $\longrightarrow$ v(I,N) + link(I,I2) + v(I2,MM) : M > N+1 , MM = N+1

Case-studies: Self-organizing networks: 3/5

- MSR specification
- **rule(Head, Body, Constraint, N).**
- **Head** and **Body**: are Prolog lists of terms with free variables or empty list.
- **Constraint**: is a CLP constraints $\{C_1;\ldots;C_n\}$ or {} (true) where C_i is a relation over variables occurring in Head and Body
- **N**: is constant used as identifier of the rule

Case-studies: Self-organizing networks: 4/5

- Unsafe states are represented via the fact **unsafe ([U_1; ...;U_k])**. Where U_i is an unsafe term(Multiset,Constraint).
- Invariants have a similar structure with unsafe terms replaced by **inv([...])**. Invariants are used to prune the backward search: all constrained multisets subsumed by an invariant are removed at each step of the search.

Case-studies: Self-organizing networks: 5/5

```
rule([init], [v(0,N),link(0,1),v(1,M),link(1,2),
              v(2,M),link(2,3),v(3,M),link(3,0)],
             {N=0 , M=100},rule1).


rule([v(I,N),link(I,I2),v(I2,M)],
     [v(I,N),link(I,I2),v(I2,MM)],
     {M>N+1 , MM=N+1},rule2).


unsafe([
      %this is a list of state useful to test rule2
       unsafe([v(I,N),link(I,I2),v(I2,N2)],{N=1 ,N2=2}),
       %unsafe([v(I,N),link(I,I2),v(I2,N2)],{}),
       %unsafe([v(I,N),link(I,I2),v(I2,N2)],{N=3,N2=4}),
       %unsafe([v(I,N),link(I,I2),v(I2,N2)],{N=1,N2=1}),
       %unsafe([v(I,N),link(I,I2),v(I2,N2)],{N=2,N2=_}),
       %unsafe([v(I,N),link(I,I2),v(I2,N2)],{N=1}),
       %unsafe([v(I,N),link(I,I2),v(I2,N2)],{I=0,I2=1,N=0})
       ]).


invariants([
            inv([v(I,_),link(I,I2),v(I2,_)],{I2=I}),
            inv([v(_,N)],{N<0}),
            inv([v(0,N)],{N>0}),
            inv([v(I,N),link(I,I2),v(I2,N2)],{I2>I , N2<N})
            ]).
```

Conclusion: 1/2

- The MSR(C) prototype is an excellent tool to model-check infinite state systems.
- Defects:
  - No debug function

Conclusion: 2/2

- Future work:
  - PFS Prototype
  - Xtext for rewriting a new MSR prototype

PFS Prototype: 1/3

- PFS is a prototype for the verification of parameterized systems.

- The prototype performs backward reachability over an approximation of the system in order to check safety properties.

PFS Prototype: 2/3

- Parameterized systems: each process is a finite-state machine
- Transition within a process can be one of the following forms:
  - A process can change the values of its local variables and possibly move to another control state.
  - A process can check if all other processes (or all processes to its left/right) satisfy a given condition over their local variables. (Then proceeds like rule 1)
  - A process can check if at least one other process (or one process to its left/right) satisfies a given condition over its local variables. (Then proceeds like rule 1)

PFS Prototype: 3/3

- PFS checks parameterized system models against safety properties.
- The set of unsafe configurations corresponds to the set of sequences containing at least two process states where the control location is critical.
- The goal is to check if the set of unsafe configurations is unreachable from the set of “initial” configurations regardless of the number of processes in the system.